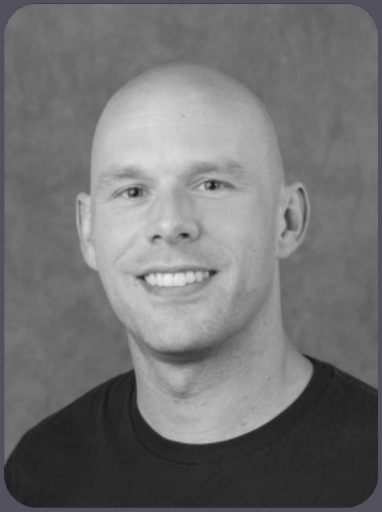


hacker bits

October 2016

new bits

Simplify and automate.

Here at *Hacker Bits*, we are big advocates of simplicity. We believe that when the going gets tough, the tough simplifies. When we are snowed under with work, we don't dream about chucking it all and running away to Hawaii (OK, we do think about that, sometimes). Rather, we simplify and automate.

And as part of our new plan to improve *Hacker Bits*, we are simplifying the design of the magazine with single columns for easy reading and scrolling. Our eventual goal is to automate the generation of the magazine so that we can speed up the delivery of information to our users. If we succeed, we'll be able to bring you *Hacker Bits* weekly instead of monthly!

We are also making bold plans to take *Hacker Bits* mobile: we'll be expanding to mobile formats like epub and mobi, so that you can *learn more*, *read less* and *stay current* wherever you go!

And speaking of staying current, don't miss this month's most highly-voted article on web scraping. Francis Kim gives us the low-down on this controversial technology and an up-to-date report on the state of web scraping in 2016.

See y'all back here in November!

— Maureen and Ray

us@hackerbits.com

content bits

October 2016

contributor bits

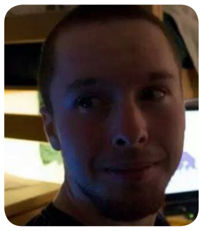
Brandon Hays
Brandon left the world of marketing to find that creating software made him happy. Brandon lives in Austin, TX, where he helps run The Frontside, a UI engineering consultancy. Brandon's lifelong mission is to hug every developer.

Heydon Pickering
Heydon is a designer, author, illustrator and migraineur. His new book *Inclusive Design Patterns* is out now. You can reach him @heydonworks.

Francis Kim
Francis is a web developer born in Gangnam, Korea, living in Melbourne, Australia. While his career's mainly been focused around eCommerce, he loves being a hacker — creating scripts and bots to automate boring, everyday tasks.

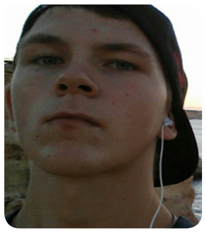
Andrey Petrov
Andrey doodles with code, tweets, and stick figures. Once upon a time, he founded a YC company and sold a company to Google. He maintains many open source projects, and you've probably used one of them. You can reach him at shazow.net or @shazow.

Matthew Jones
Matthew is a software developer for U-Haul Intl. He writes the blog Exception Not Found where he attempts to be funny and insightful, and sometimes even manages it. He lives in Phoenix with his wife and three kids.



Alexander Kyte
Alexander is a student in his last year in Computer Science at the Rochester Institute of Technology. He works for Microsoft on the Mono C# virtual machine.

John De Goes
John is an entrepreneur, functional programmer, open source contributor, speaker, author, and science addict. CTO at SlamData. Work hard, stay positive, live fearlessly. You can reach him @jdegoes.



Illya Gerasymchuk
Illya is a student currently getting his Masters degree in Software Engineering and Distributed Systems. His interests span from low-level to web development. You can find some of the projects he's working on on his GitHub: iluxonchik.

Ulas Türkmen

Ulas has been working as a developer for nearly ten years, after studying engineering and cognitive science, and then settling down as a web developer. He spends his time mostly in the Python world, with regular excursions to Elisp and Haskell.

Arthur Levy

Formerly in investment banking and private equity, Arthur is now Head of Partnerships at [Teespring](), an eCommerce platform that has raised over $60M from YC, Andreessen Horowitz and Khosla Ventures since 2011 launch.

Ray Li

Curator

Ray is a software engineer and data enthusiast who has been blogging at [rayli.net]() for over a decade. He loves to learn, teach and grow. You'll usually find him wrangling data, programming and lifehacking.

Maureen Ker

Editor

Maureen is an editor, writer, enthusiastic cook and prolific collector of useless kitchen gadgets. She is the author of 3 books and 100+ articles. Her work has appeared in the *New York Daily News*, and various adult and children's publications.

The conjoined triangles of senior-level development

By BRANDON HAYS

"This actually makes me feel less confident about my role here. If we can't define what we think senior is, how am I supposed to know if I'm working toward it?"

At Frontside, we gather every Tuesday afternoon for our company meeting, where we talk through our accomplishments and plans for the coming week.

In a recent meeting, we talked about our search for a senior developer to join the team, and you could see passions flare. No decision we can make impacts the company quite like bringing on a new team member, so naturally we all had strong opinions about the qualifications we were looking for.

However, *not one person* in the room could crystallize our thoughts on what qualifies as "senior" above a general gut feeling.

One person chimed in with the quote above, and I realized that we'd stumbled onto one of the biggest problems in our entire company: *we had no idea how to define the role we're trying to hire for and grow our developers toward.*

The problem of defining "Senior Developer"

For my part, I've come to distrust the term "Senior Developer", particularly after being given that title after 9 months of professional programming experience to fit into a salary band.

In fact, *if you ask two experienced developers to specify their definition of "senior", I guarantee you will find serious conflicts between their answers.*

The question of what qualifies as a "Senior Developer" is both context-dependent and so malleable that our industry squishes it into whatever shape fits their current needs.

Here are some actual, real-world definitions I've seen used to justify the label "Senior Developer"

- 15+ years of programming experience
- 2 years of programming experience and decent leverage
- 1 year of experience in a hot new 1-year-old framework
- Having authored a technical book
- Writing Comp Sci algorithms on a whiteboard
- Writing an open source library used within the company

That's a pretty broad range of definitions. But lots of stuff in life is hard to define, so what's the problem?

Why bother defining it? What's wrong with a gut feeling?

In our meeting, when our team member pointed out their confusion, they were pointing out that we are hiring, firing, and promoting people based on criteria we can't define or defend. They're right, that is *bananas*.

Worse, we were *failing at our core mission* of growing developers by not clearly marking the path we're trying to help them travel.

There's another, larger problem with "I'll know a senior developer when I see one": *"Senior Developer" is a profoundly effective vector for bias.*

“Senior Developer” as a way to enshrine bias

When we picture a wise senior developer, we each have our own experiences and preferences that color this picture in for us. That means the term is already spring-loaded with a bunch of our own baggage.

When we take a “gut feeling” sense of someone’s seniority without specific criteria, there is basically no way to counteract our own biases, but we still make a judgement. *It’s completely possible for a person applying to multiple dev jobs to be evaluated as junior at one, mid-level at another, and even senior at another, with very little feedback as to why.*

And as hiring managers, we all feel correct when making these judgements, even as we reach vastly different conclusions.

The result is a reinforcement of existing biases that keeps some people from progressing, while resulting in “title inflation” for others. Since *existing biases in tech are strongly and undeniably tilted toward white males*, the existing “gut feeling” system disproportionately harms women and people of color.

Why haven’t we solved this?

This definition is an extremely hard problem, since it’s heavily dependent on the context of the work environment. Most business owners are just making this stuff up as they go along, and the solution we tend to wind up with seems to be “good enough”.

There’s also no incentive, because *defining criteria takes a lot of the “gut decision” power away from founders/owners and adds accountability.* Who sits around asking for less power and more accountability for themselves?

Injecting accountability

I like accountability. I take comfort in it. I’ve learned that being held accountable for something is quite liberating. For instance, being clear about the criteria we use to hire someone leads to less regret than “going with my gut”, *which can be affected by everything from someone’s ability to riff on Simpsons quotes with me to whether I skipped breakfast that day.*

Accountability also opens the door for improvement. As the hiring manager, I’m responsible for fostering a safe, capable, happy, diverse team. Improving and working toward those goals can either happen by gut feelings and dumb luck, or we can define, measure, and hold ourselves accountable for iterating toward those goals.

Accountability moves us from being passengers to drivers of our own future.

It starts with responsibilities

The question becomes: *how can we create and use quantifiable criteria for seniority without creating a deeply flawed, highly game-able system?*

The only fair way I can think of to judge a candidate is by asking a few key questions: *what are this person’s responsibilities? How likely are they able to fulfill them? How much help will they need?*

We started by defining our environment. When we examine the key attributes of the environment at Frontside, it starts to become clear:

- Being small means that people need to take the lead in several roles to solve problems from start to finish. There are no “cogs” in this machine.
- We rely and build on the strength of our internal community and the larger communities we participate in, particularly in open source.
- We have extremely ambitious technical goals and standards for the maintainability and usability of the code we write.

The resulting responsibilities are then pretty easy to determine:

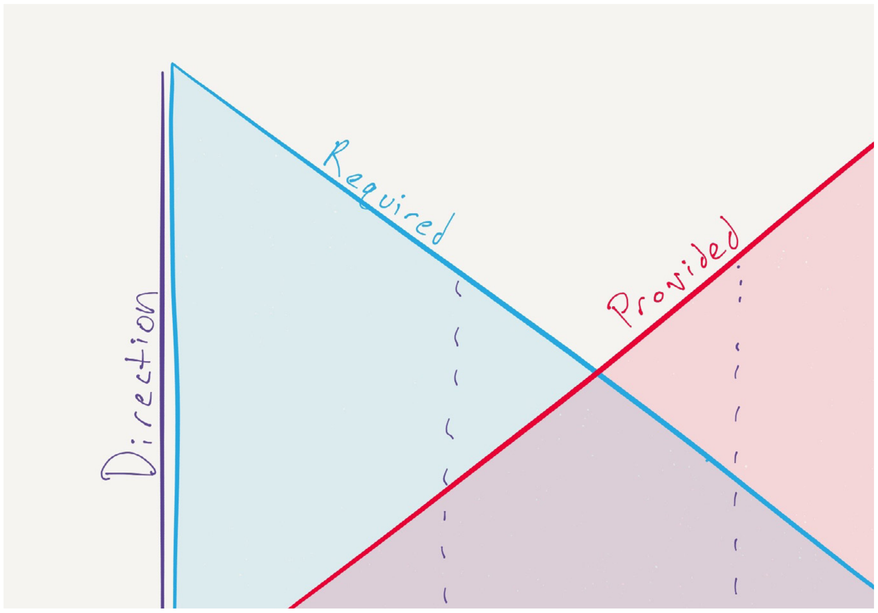
- Provide clear technical and project guidance for their teammates and our clients
- Mentor, teach, and contribute internally and in larger programming communities
- Ship software that is a joy to its users and to the developers who extend and maintain it

Together, these responsibilities form the basis of the criteria we use to assess seniority.

The Conjoined Triangles: the simple explanation

I recently had a chance to dig into the definition of “senior” at a number of companies, large and small, and came away with only one common denominator.

The simplest explanation of seniority across companies is this: *how much direction will this person need, and how much will they be able to provide to others?*



I stand by the Conjoined Triangles of Senior-Level Development as a nice idea, but like Action Jack’s “Conjoined Triangles of Success”, it’s enough of an oversimplification to shed its intrinsic meaning.

Even with this understanding, the door is wide open to inject bias, miss important criteria, and overvalue higher-visibility things like public notoriety or the ability to speak computer science lingo.

In the meeting, a new framework emerges

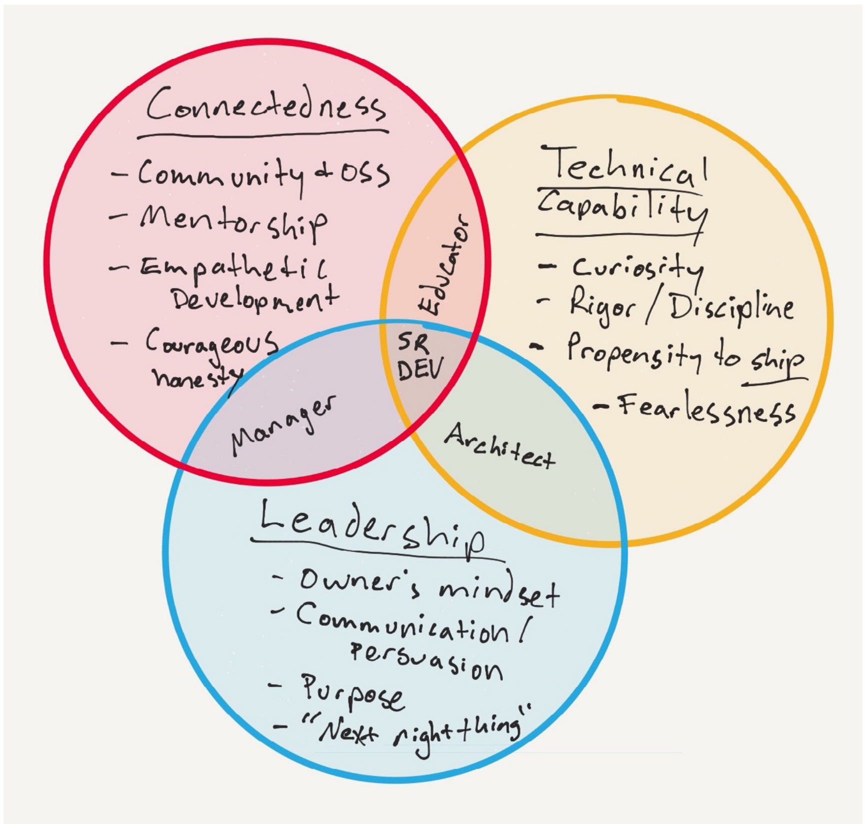
A really cool thing happened during the heated meeting. As I was describing this simple “unified field theory”, another employee put forward a new mental model to organize our thinking around this.

She described the framework we use to determine seniority at Frontside as a Venn diagram, the intersection of how well a person works independently and leads, how technically capable they are, and how well they connect and contribute to a larger community.

The Venn diagram: the more complex explanation

Our evaluation of seniority does, in fact, roll up to the higher definition: *"How much direction will this person need, and how much will they be able to provide to others?"* But as our employees pointed out, when we stop there, there's a ton of room for confusion.

So how can we decide how well a candidate is likely to perform their responsibilities? *How do we anchor our evaluation criteria to something concrete, without killing it by turning it into a mathematical formula?*



When we boiled down what we're looking for, we came away with 12 traits that divide pretty cleanly along those three areas of responsibility: *technical capability*, *leadership*, and *community*.

I'll break the 12 traits down in detail in the next post, but for now, we'll just hit the high points.

The tracks

Technical capability: A person with high technical capability is technically curious, tackles problems without giving up, and produces solutions that less-experienced folks can use, maintain, and learn from.

Leadership: A person with leadership skills knows how to develop and follow a sense of purpose, in themselves and in others. They are willing to point out, own, and fix things that are broken about our company and in their own career tracks.

Community/Connectedness: A person with community skills has a sense of being part of a larger whole, a desire to contribute, a sense that the other people (i.e. coworkers, users, clients) are not simply characters in his or her own movie, but fully-realized individuals.

What about "culture fit"?

We initially almost called the Community & Connectedness track "culture fit", but I have come to be highly suspicious that this term is a thought-terminating cliché. *"Culture fit" is a junk drawer* for all the stuff you want to see in a developer that you can't otherwise define, and it's an ideal hiding place for intrinsic biases.

When we defined the criteria that make Frontside's culture unique, the common trait is the idea of connectedness as defined above.

Measuring seniority across 3 disparate tracks

Remember the three areas, *technical capability*, *leadership*, and *community*? Each of these tracks has a junior-to-senior path all its own.

It's now common for a person to switch careers and come out of a code bootcamp with seniority in leadership and connectedness, while being junior in technical capability. Conversely, a skilled, formally-trained technician may be missing experience in both leadership and connectedness.

Very few qualify as senior in all three categories. In fact, *it's unlikely a person will even be interested* in becoming senior in all three categories. At Frontside, we define seniority as a blend of these tracks and work to help people level up where they feel like they want to improve.

Artifacts: the only measuring tape you've got

The measure of seniority in each track requires evidence. If you've done something, you likely have some kind of artifact as a result.

The 12 traits we'll discuss in the next post each come with concrete artifacts a candidate can point to show that yes, they have developed those traits over time and with practice.

But taken as a whole, *a couple of well-developed and deeply-practiced traits in an area are likely to qualify a person as senior in that area.*

For instance, if a person can talk you through the code they've written across various programming languages, they probably peg the "curiosity" trait pretty high. Combine that with the rigor to write thorough, high-quality tests for the majority of their projects and hook it up to Continuous Integration, you're probably looking at someone we'd consider technically senior.

Or, if someone has a long history of mentoring others, organizing meetups, or creating tools that make others' lives easier, it's likely we'd consider them senior on the community track.

If a person has managed a few teams before, they have likely learned and developed processes to handle the very real challenges there. Combine that with the ability to get to the purpose or root of an issue, and you have someone we'd call senior on the leadership track.

How we define "Senior"

Our measure is that if someone is senior on the technical track and either the leadership or community track, they're senior to us. We can help level them up on the remaining track if they want.

If they're senior on the community and leadership side, we consider them senior if they are high-mid-level on the technical track.

As a real-life example: about a year ago, we hired a new employee as a junior developer, because we knew on the technical side they needed a ton of direction for at least the first six months.

At six months, they were solidly mid-level technically, and within their first year, by our estimation, they were already senior. I can say this with confidence because when they left, we realized that to fulfill their job functions, we'd need to hire a senior developer.

This happened because when they started with us, they were already senior on the community and leadership tracks, so *all they needed was a boost on the technical side* to perform the job of a senior dev on our team.

Technical skills aren't enough

For someone with technical skills but little experience in community or leadership, *"You don't qualify as senior by our standards"* is not music to their ears. But for the job responsibilities they'll have with us, we would consider that person mid-level until we can help strengthen one or both of the remaining tracks for them.

Many shops only measure along the technical axis, but it simply doesn't work for a company of our small size and collaborative work style. I also worry that those who only measure technical skill are buying into the dangerous "lone genius developer" myth, cheating the developer out of the serious boost they'd gain by adding leadership or community-oriented skills to their technical savvy.

Even in larger companies with more focused roles, *I'd like to see a broader definition of "Senior Developer" that encompasses the code and non-code skills we use,* and that allow us to work effectively, particularly in teams and with real customers.

How long does it take to become senior?

Does "Senior Developer" mean "X years of experience"? Well, I don't see it happening inside of 5 years for anyone. It would be difficult or impossible to develop the traits needed to reach senior-level in any of these tracks in less than several years, much less along multiple tracks.

But "5 years of experience" doesn't necessarily mean "5 years of software development experience". A person who has already climbed the Leadership and/or Connectedness tracks need only hook that up to their increasing technical skills to provide a tremendous amount of value as a software developer.

Our hiring "secret sauce" largely stems from the fact that *it seems to take significantly less time for someone with leadership and community skills to develop technical skills than the other way around.* I'm seeing a large number of people who graduated from code bootcamps 3 and even 2 years ago now handily and gracefully filling the role of senior developer.

There's more to discuss

This leaves many questions unanswered. What are the specific artifacts we use to evaluate skills and traits in the three areas? How can you evaluate for these before and during an interview? How do we tie these evaluations to something as concrete as someone's salary?

How does this framework apply to non-senior developers? How and when do developers advance? What's the difference between a Junior, Mid, and Senior? What exists between them? *Are these terms devoid of meaning and in need of replacement?*

Lastly, how does this framework apply (if it applies at all) to other companies with vastly different cultures and needs from Frontside?

In the next post, we'll dive into specifics to answer these.

Moving past "gut feelings"

Defining "senior" is an ongoing and surprisingly difficult process, but we do it because it's business-critical for us. Without a clear definition of "senior developer", we have no clear path for our own employees to get there. We have no concrete way to evaluate people joining the team, no way to hold ourselves accountable, and no way of improving the process.

As an industry, it's time to graduate from the "I'll know it when I see it" definition toward something we can define and share. *Let's bring open source thinking to how we hire and grow our people.*

I hope to answer the major lingering questions in the next post. If you have questions or a different definition, you can find me on Twitter (@tehviking) or better yet, write a post with your own thoughts and I'll link to it. ■

Reprinted with permission of the original author. First appeared at frontside.io.

Writing less code

By HEYDON PICKERING

I'm not the most talented coder in the world. No, it's true. So I try to write as little code as possible. The less I write, the less there is to break, justify, or maintain.

I'm also lazy, so it's all gravy. (*Ed: maybe run with a food analogy?*)

But it turns out the only surefire way tomake performant Web Stuff is also to just write less. Minify? Okay. Compress? Well, yeah. Cache? Sounds technical. Flat out refuse to code something or include someone else's code in the first place? *Now you're talking.*

What goes in one end has to come out the other in some form, whether it's broken down and liquefied in the gastric juices of your task runner or not. (*Ed: I've changed my mind about the food analogy.*)

And that's not all. Unlike aiming for 'perceived' performance gains — where you still send the same quantity of code but you chew it up first (*Ed: seriously*) — you can actually make your Web Stuff *cheaper* to use. My data contract doesn't care whether you send small chunks or one large chunk; it all adds up the same.

My *favorite* thing about aiming to have less stuff is this: you finish up with only the stuff you really need — only the stuff your *user* actually wants.

Massive hero image of some dude drinking a latte? Lose it. Social media buttons which pull in a bunch of third-party code while simultaneously wrecking your page design? Give them the boot. That JavaScript thingy that hijacks the user's right mouse button to reveal a custom modal? Ice moon prison.

It's not just about what you pull in to destroy your UX or not, though. The *way* you write your (own) code is also a big part of having less of it.

Here are a few tips and ideas that might help. I've written about some of them before, but in terms of accessibility and responsive design. It just happens that a flexible, accessible Web is one we try to exert little of our own control over; one we do less to break.

WAI-ARIA

First off, WAI-ARIA != web accessibility. It's just a tool to enhance compatibility with certain assistive technologies, like screen readers, where it's needed. Hence, the first rule of ARIA use is to *not* use WAI-ARIA if you don't have to.
LOL, no:

```
<div role="heading" aria-level="2">Subheading</div>
```

Yes:

```
<h2>Subheading</h2>
```

The benefit of using native elements is that you often don't have to script your own behaviors either. Not only is the following checkbox implementation verbose HTML, but it needs a JavaScript dependency to control state changes and to follyfill standard, basic behavior regardingthe `name` attribute and `GET` method. It's more code, and it's less robust. Joy!

```
<div role="checkbox" aria-checked="false" tabindex="0" id="checkbox1"
     aria-labelledby="label-for-checkbox1"></div>
<div class="label" id="label-for-checkbox1">My checkbox label</div>
```

Styling? Don't worry, you're covered. That's if you really need custom styles, anyway.

```
<input type="checkbox" id="checkbox1" name="checkbox1">
<label for="checkbox1">My checkbox label</label>
```

Grids

Do you remember ever enjoying using/reading a website with more than two columns? I don't. Too much stuff all at once, begging for my attention. "I wonder which thing that looks like navigation is the navigation I want?” It’s a rhetorical question: my executive functioning has seized up and I’ve left the site.

Sometimes we want to put things next to things, sure. Like search results or whatever. But why pull in a whole ton of grid framework boilerplate just for that? Flexbox can do it with no more than a couple of declaration blocks.

```
.grid {
  display: flex;
  flex-flow: row wrap;
}

.grid > * {
  flex-basis: 10em;
  flex-grow: 1;
}
```

Now everything ‘flexes’ to be approximately `10em` wide. The number of columns depends on how manyapprox’ `10em` cells you can fit into the viewport. Job done. Move on.

Oh and, while we’re here, we need to talk about this kind of thing:

```
width: 57.983635273564737827364645463733373737%;
```

Did you know that precise measurement is calculated according to a mystical ratio? A ratio which purports to induce a state of calm and awe? No, I wasn’t aware and I’m not interested. Just make the porn button big enough that I can find it.

Margins

We’ve done this. Share your margin definition across elements using the universal selector. Add overrides only where you need them. You won’t need many.

```
body * + * {
  margin-top: 1.5rem;
}
```

No, the universal selector will not kill your performance. That’s bunkum.

Views

You don’t need the whole of Angular or Meteor or whatever to divide a simple web page into ‘views’. Views are just bits of the page you see while other bits are unseen. CSS can do this:

```
.view {
  display: none;
}

.view:target {
  display: block;
}
```

“But single-page apps run stuff when they load views!” I hear you say. That’s what the onhashchange event is for. No library needed, and you’re using links in a standard, bookmark-able way. Which is nice. There’s more on this technique, if you’re interested.

Font sizes

Tweaking font sizes can really bloat out your `@media` blocks. That's why you should let CSS take care of it for you. With a single line of code.

```
font-size: calc(1em + 1vw);
```

Err… that's it. You even have a minimum font size in place, so no tiny fonts on handsets. Thanks to Vasilis for showing me this.

10k Apart

Like I said, I'm not the best coder. I just know some tricks. But it is possible to do a hell of a lot, with only a little. That's the premise of the 10k Apart competition — to find out what can be made with just 10k or less. There are big prizes to be won and, as a judge, I look forward to kicking myself looking at all the amazing entries; ideas and implementations I wish I'd come up with myself. What will you make? ■

Web scraping in 2016

By FRANCIS KIM

Social media APIs and their rate limits have not been nice to me recently, especially Instagram. Who needs it anyway?

Sites are increasingly getting smarter against scraping/data mining attempts. AngelList even detects PhantomJS (I have not seen other sites do this). But if you are automating your exact actions that happen via a browser, can this be blocked?

First off, in terms of concurrency or the amount of horsepower you get for your hard earned $$$ — Selenium sucks. It's simply not built for what you would consider 'scraping'. But with sites being built with more and more smarts these days, the only truly reliable way to mine data off the internets is to use browser automation.

My stack looks like, pretty much all JavaScript. There goes a few readers — WebdriverIO, Node.js and a bunch of NPM packages including the likes of antigate (thanks to Troy Hunt's "Breaking CAPTCHA with automated humans"), but I'm sure most of my techniques can be applied to any flavour of the Selenium 2 driver. It just happens that I find coding JavaScript optimal for browser automation.

Purpose of this post

I'm going to share everything that I've learnt to date from my recent love affair with Selenium automation/scraping/crawling. The purpose of this post is to illustrate some of the techniques I've created that I haven't seen published anywhere else — as a broader, applicable idea to be shared around and discussed by the webdev community.

Scraping? Isn't it illegal?

(Note: the following section is taken from my Hacker News comment.)

On the ethics side, I don't scrape large amounts of data — e.g. giving clients lead gen (x leads for y dollars) — in fact, I have never done a scraping job and don't intend to do those jobs for profit.

For me it's purely for personal use and my little side projects. I don't even like the word scraping because it comes loaded with so many negative connotations (which sparked this whole comment thread) — and for a good reason — it's reflective of the demand in the market. People want cheap leads to spam, and that's bad use of technology.

Generally I tend to focus more on words and phrases like 'automation' and 'scripting a bot'. I'm just automating my life, and I'm writing a bot to replace what I would have to do on a daily basis — like looking on Facebook for some gifs and videos then manually posting them to my site. Would I spend an hour each and every day doing this? No, I'm much lazier than that.

Who is to tell me what I can and can't automate in my life?

Let's get to it.

Faking human delays

It's definitely good practice to add these human-like, random pauses in some places just to be extra safe.

```
const getRandomInt = (min, max) => {
    return Math.floor(Math.random() * (max - min + 1)) + min
}

browser
    .init()
    // do stuff
    .pause(getRandomInt(2000, 5000))
    // do more stuff
```

Parsing data jQuery style with Cheerio

Below is a snippet from a function that gets videos from a Facebook page.

```
fastFeed.parse(data, (err, feed) => {
    if (err) {
        console.error('Error with fastFeed.parse() - trying via Selenium')
        console.error(err)
        browser
            .url(rssUrl)
            .pause(10000)
            .getSource()
            .then((source) => {
                source = source.replace(/&lt;/g, '<')
                               .replace(/&gt;/g, '>').replace(/&amp;/g, '&')
                source = source.replace(/(<.?html([^>]+)*>)/ig, '')
                               .replace(/(<.?head([^>]+)*>)/ig, '')
                               .replace(/(<.?body([^>]+)*>)/ig, '')
                               .replace(/(<.?pre([^>]+)*>)/ig, '')
                if (debug) console.log(source)
                fastFeed.parse(source, (err, feed) => {
                    // let's go further up the pyramid of doom!
                }
```

I also use this similar method and some regex to parse RSS feeds that can't be read by command line, cURL-like scripts.

It *actually* works pretty well — I've tested with multiple sources.

Injecting JavaScript

If you get to my level, injecting JavaScript for the client-side becomes commonplace.

By the way, this is a totally non-practical example (in case you haven't noticed). Check out the following headings.

```
browser
    // du sum stufs lel
    .execute(() => {
        let person = prompt("Please enter your name", "Harry Potter")
        if (person != null) {
            alert(`Hello ${person}! How are you today?`)
        }
    })
```

Beating CAPTCHA

GIFLY.co had not updated for over 48 hours and I wondered why. My script, which gets animated gifs from various Facebook pages, was being hit with the capture screen.

Cracking Facebook's captcha was actually pretty easy. It took me exactly 15 minutes to accomplish this. I'm sure there are ways to do this internally but with Antigate providing an NPM package and with costs so low, it was a no-brainer for me.

```
const Antigate = require('antigate')
let ag = new Antigate('booo00000haaaaahaaahahaaaaaa')

browser
    .url(url)
    .pause(5000)
    .getTitle()
    .then((title) => {
        if (!argv.production) console.log(`Title: ${title}`)
        if (title == 'Security Check Required') {
            browser
                .execute(() => {
                    // injectz0r the stuffs necessary
                    function convertImageToCanvas(image) {
                        var canvas = document.createElement("canvas")
                        canvas.width = image.width
                        canvas.height = image.height
                        canvas.getContext("2d").drawImage(image, 0, 0)
                        return canvas
                    }
                    // give me a png with base64 encoding
                    return convertImageToCanvas(
                        document.querySelector('#captcha img[src*=captcha]')).toDataURL()
                })
                .then((result) => {
                    // apparently antigate doesn't like the first part
                    let image = result.value.replace('data:image/png;base64,', '')
                    ag.process(image, (error, text, id) => {
                        if (error) {
                            throw error
                        } else {
                            console.log(`Captcha is ${text}`)
                            browser
                                .setValue('#captcha_response', text)
                                .click('#captcha_submit')
                                .pause(15000)
                                .emit('good') // continue to do stuffs
                        }
                    })
                })
        }
```

So injecting JavaScript has become super handy here. I'm converting an image to a canvas, then running `.toDataURL()` to get a Base64 encoded PNG image to send to the Antigate endpoint. The function was stolen from a site where I steal a lot of things from, shout outs to David Walsh. This solves the Facebook captcha, enters the value then clicks submit.

Catching AJAX errors

Why would you want to catch client-side AJAX errors? Because reasons. For example, I automated unfollowing *everyone* on Instagram and I found that even through their website (not via the API) there is some kind of a rate limit.

```
browser
    // ... go to somewhere on Instagram
    .execute(() => {
        fkErrorz = []
        jQuery(document).ajaxError(function (e, request, settings) {
            fkErrorz.push(e)
        })
    })
    // unfollow some people, below runs in a loop
        browser
            .click('.unfollowButton')
            .execute(() => {
                return fkErrorz.length
            })
            .then((result) => {
                let errorsCount = parseInt(result.value)
                console.log('AJAX errors: ' + errorsCount)
                if (errorsCount > 2) {
                    console.log('Exiting process due to AJAX errors')
                    process.exit()
                    // let's get the hell outta here!!
                }
            })
```

Because a follow/unfollow invokes an AJAX call, and being rate limited would mean an AJAX error, I injected an AJAX error capturing function then saved it to a global variable.

I retrieve this value after each unfollow and terminate the script if I get 3 errors.

Intercepting AJAX data

While scraping/crawling/spidering Instagram, I ran into a problem. A tag page did not give me the post date in the DOM. I really needed this data for IQta.gs and I couldn't afford visiting every post as I'm parsing about 200 photos every time.

What I did find though, is that there is a `date` variable stored in the post object that the browser receives. Heck, I ended up not even using this variable but this is what I came up with.

```
browser
    .url(url) // an Instagram tag page eg. https://www.instagram.com/explore/tags/coding/
    .pause(10000)
    .getTitle()
    .then((title) => {
        if (!argv.production) console.log(`Title: ${title}`)
    })
    .execute(() => {
        // override AJAX prototype & hijack data
        iqtags = [];
        (function (send) {
            XMLHttpRequest.prototype.send = function () {
                this.addEventListener('readystatechange', function () {
                    if (this.responseURL == 'https://www.instagram.com/query/'
                        && this.readyState == 4) {
                        let response = JSON.parse(this.response)
                        iqtags = iqtags.concat(response.media.nodes)
                    }
                }, false)
                send.apply(this, arguments)
            }
        })(XMLHttpRequest.prototype.send)
    })
    // do some secret awesome stuffs
    // (actually, I'm just scrolling to trigger lazy loading to get moar data)
    .execute(() => {
        return iqtags
    })
    .then((result) => {
        let nodes = result.value
        if (!argv.production) console.log(`Received ${nodes.length} images`)

        let hashtags = []
        nodes.forEach((n) => {
            // use regex to get hashtags from captions
        })
        if (argv.debug > 1) console.log(hashtags)
```

It’s getting a little late in Melbourne.

Other smarts

So I run all of this in a Docker container running on AWS. I’ve pretty much made my Instagram crawlers fault-tolerant with some bash scripting *(goes to check if it is running now).*

```
==> iqtags.grid2.log <==
[2016-08-23 15:01:40][LOG] There are 1022840 images for chanbaek
[2016-08-23 15:01:41][LOG] chanbaek { ok: 1,
  nModified: 0,
  n: 1,
  upserted: [ { index: 0, _id: 57bc654f1007e86f09f70b49 } ] }
[2016-08-23 15:01:51][LOG] Getting random item from queue
[2016-08-23 15:01:51][LOG] Aggregating related tags from a random hashtag in db
[2016-08-23 15:01:51][LOG] Hashtag #spiritualfreedom doesn't exist in db
[2016-08-23 15:01:51][LOG] Navigating to https://www.instagram.com/explore/tags/spiritualfreedom
[2016-08-23 15:02:05][LOG] Title: #spiritualfreedom • Instagram photos and videos

==> iqtags.grid3.log <==
[2016-08-23 15:00:39][LOG] Navigating to https://www.instagram.com/explore/tags/artist
[2016-08-23 15:00:56][LOG] Title: #artist • Instagram photos and videos
[2016-08-23 15:01:37][LOG] Received 185 images
[2016-08-23 15:01:37][LOG] There are 40114945 images for artist
[2016-08-23 15:01:37][LOG] artist { ok: 1, nModified: 1, n: 1 }
[2016-08-23 15:01:47][LOG] Getting random item from queue
[2016-08-23 15:01:47][LOG] Aggregating related tags from a random hashtag in db
[2016-08-23 15:01:47][LOG] Hashtag #bornfree doesn't exist in db
[2016-08-23 15:01:47][LOG] Navigating to https://www.instagram.com/explore/tags/bornfree
[2016-08-23 15:02:01][LOG] Title: #bornfree • Instagram photos and videos
[2016-08-23 15:02:44][LOG] Received 183 images
[2016-08-23 15:02:44][LOG] There are 90195 images for bornfree
[2016-08-23 15:02:45][LOG] bornfree { ok: 1,
  nModified: 0,
  n: 1,
  upserted: [ { index: 0, _id: 57bc658f1007e86f09f70b4c } ] }
```

Yes, it seems that all 6 IQta.gs crawlers are running fine I've run into some issues with Docker where an image becomes unusable, I have no idea why — I did not spend time to look into the root cause, but basically my bash script will detect non-activity and completely remove and start the Selenium grid again from a fresh image.

Random closing thoughts

I had this heading written down before I started writing this post and I have forgotten the random thoughts I had back then. Oh well, maybe it will come back tomorrow.

Ah, a special mention goes out to Hartley Brody and his post, which was a very popular article on Hacker News in 2012/2013 — it inspired me to write this.

Those of you wondering what the hell `browser` is:

```
const webdriverio = require('webdriverio')
let options
if (argv.chrome) {
    options = {
        desiredCapabilities: {
            browserName: 'chrome', chromeOptions: {
                prefs: {
                    'profile.default_content_setting_values.notifications': 2
                }
            }
        },
        port: port,
        logOutput: '/var/log/fk/iqtags.crawler/'
    }
}
else {
    options = {
        desiredCapabilities: {
            browserName: 'firefox'
        },
        port: port,
        logOutput: '/var/log/fk/iqtags.crawler/'
    }
}
const browser = webdriverio.remote(options)
```

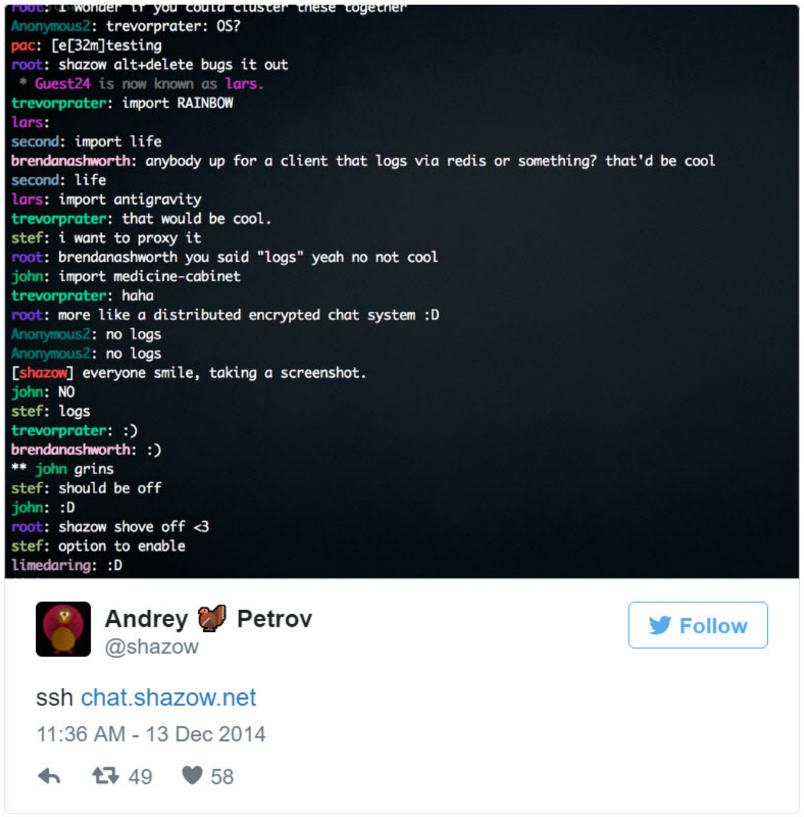
And `argv` comes from yargs.

Thanks to my longtime friend In-Ho @ Google for proofreading! You can check out some of his music production here. ■

Why aren't we using SSH for everything?

By ANDREY PETROV

A few weeks ago, I wrote ssh-chat.



```
root: I wonder if you could cluster these together
Anonymous2: trevorprater: OS?
pac: [e[32m]testing
root: shazow alt+delete bugs it out
 * Guest24 is now known as lars.
trevorprater: import RAINBOW
lars:
second: import life
brendanashworth: anybody up for a client that logs via redis or something? that'd be cool
second: life
lars: import antigravity
trevorprater: that would be cool.
stef: i want to proxy it
root: brendanashworth you said "logs" yeah no not cool
john: import medicine-cabinet
trevorprater: haha
root: more like a distributed encrypted chat system :D
Anonymous2: no logs
Anonymous2: no logs
[shazow] everyone smile, taking a screenshot.
john: NO
stef: logs
trevorprater: :)
brendanashworth: :)
** john grins
stef: should be off
john: :D
root: shazow shove off <3
stef: option to enable
limedaring: :D
```

Andrey Petrov
@shazow

Follow

ssh chat.shazow.net

11:36 AM - 13 Dec 2014

49 58

The idea is simple: You open your terminal and type,

```
$ ssh chat.shazow.net
```

Unlike many others, you might stop yourself before typing “ls” and notice—that’s no shell, it’s a chat room!

While the little details sink in, it dawns on you that there is something extra-special going on in here.

SSH knows your username

When you ssh into a server, your client shares several environment variables as inputs to the server, among them is the $USER variable. You can overwrite this, of course, by specifying the user you’re connecting as such:

```
$ ssh neo@chat.shazow.net
```

Well look at that, you’re the one. So special. What else can we mess with? By default, the server gets your $TERM variable too.

```
$ TERM=inator ssh chat.shazow.net
```

If ssh-chat was smart, it would recognize that your custom terminal might not support colours and skip sending those extra formatting characters.

You can even push your own environment variables with a *SendEnv* option flag, but we won't get into that.

SSH validates against your key pair

There are several supported authentication methods in SSH: *none*, *password, keyboard-interactive,* and *publickey*. They each have interesting properties, but the last one is especially handy.

When your SSH client connects to a server, it negotiates an acceptable authentication method that both the client and server support (typically the reverse order of the above). If you've specified the identity flag or have some keys lying around in your *~/.ssh/* directory, your client will offer up some public keys to authenticate against.

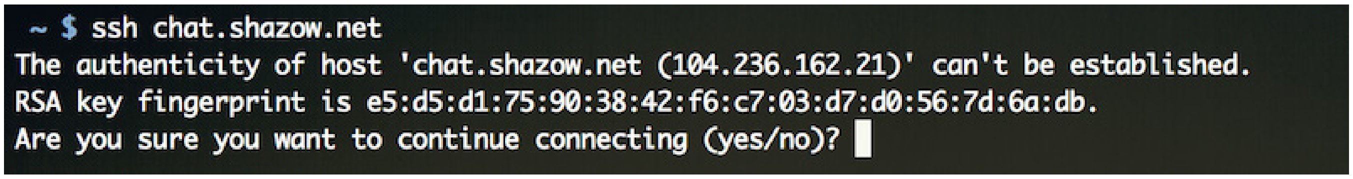
If the server recognizes one of those keys, such as if they're listed in *authorized_keys*, then a secure handshake will begin verifying that you hold the private key to that public key but without revealing what the private key is. In the process, the client and server securely agree on a temporary symmetric *session key* to encrypt the communication with.

What does this mean? It means that SSH has authentication built into the protocol. *When you join ssh-chat, not only do I know who you claim to be, but I can also permanently and securely attach an identity to your connection without any user intervention on your part.* No signup forms, no clicking on links in your email, no fancy mobile apps.

A future version of ssh-chat will allow you to create a permanent account that is validated against your key pair, and these permanent accounts might have all kinds of special features like username ownership, always-online presence, and push notifications over Pushover or email.

SSH connections are encrypted

The server uses the same kind of key pair as a client would. When you connect to a new SSH host, you get a message that presents a "key fingerprint" for you to validate. The fingerprint is the hex of a hash of the server's public key.

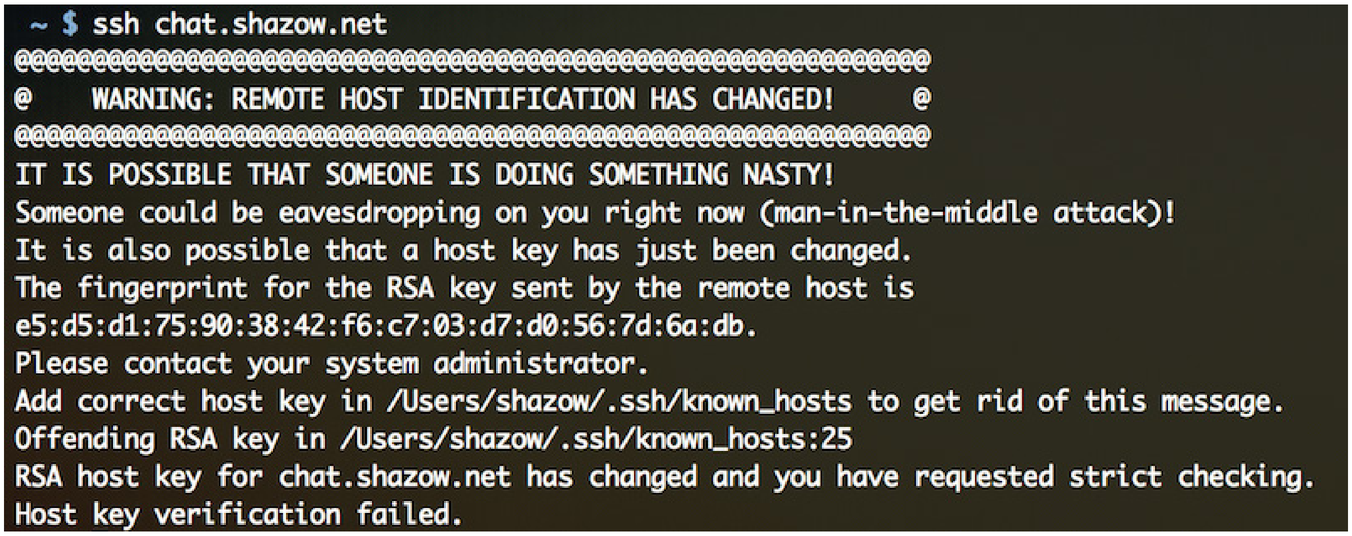


```
 ~ $ ssh chat.shazow.net
The authenticity of host 'chat.shazow.net (104.236.162.21)' can't be established.
RSA key fingerprint is e5:d5:d1:75:90:38:42:f6:c7:03:d7:d0:56:7d:6a:db.
Are you sure you want to continue connecting (yes/no)?
```

What does it mean if you try to connect to chat.shazow.net and you see a *different* fingerprint hash? You're being man-in-the-middle'd.

Your local neighbourhood clandestine security agency could make an SSH server that just acts as a proxy in front of another SSH server you frequent (using something like sshmitm) and log everything that is going on while passing it through. Fortunately, as long as the proxy doesn't have the original server's private key, then the key fingerprint will be different.

Once you accept a fingerprint, it will be added to your *~/.ssh/known_hosts* where it will be *pinned* to that host. This means if the key for the host ever changes, you'll be greeted with this appropriately-scary message:


```
 ~ $ ssh chat.shazow.net
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
e5:d5:d1:75:90:38:42:f6:c7:03:d7:d0:56:7d:6a:db.
Please contact your system administrator.
Add correct host key in /Users/shazow/.ssh/known_hosts to get rid of this message.
Offending RSA key in /Users/shazow/.ssh/known_hosts:25
RSA host key for chat.shazow.net has changed and you have requested strict checking.
Host key verification failed.
```

A host you've connected to previously is advertising a different public key than it did before. If you can't account for this change (maybe you launched a new VPS on the same IP address as before and it generated a fresh SSH key pair?) then it's worth being worried.

Try connecting to this host from another network, see if the problem persists—if not, then someone is hijacking your local connection rather than the server's connection.

SSH supports multiplexing

When your client connects to a server, it opens a *channel* where it requests a specific feature. There are many fun things your client can request like *pty-req* (a pseudo-terminal), *exec* (run command), or even *tcpip-forward* (port forwarding). There are many others, and there is nothing stopping you from inventing your own type for a custom client/server implementation. Maybe we'll see a *chat* channel someday?

The best part is that you can do all of these things concurrently: start port forwarding while opening a shell while having some command run in the background.

Once your pipeline is open, you can send *more* commands within it. When your client opens a *pty-req*, it sends event updates such as *window-change* whenever your terminal size changes.

SSH is ubiquitous

"Is it mobile-friendly?" you may joke, but it is! Every platform you can imagine, there is an SSH client available, including iOS, Android, even Windows! OSX and every Linux distro ships with a client. There are even browser extension SSH clients.

SSH is one of the most accessible secure protocols ever, second only to HTTPS of course.

SSH sounds awesome, and familiar…

Let's see what we have so far: binary protocol, mandatory encryption, key pinning, multiplexing, compression (yes, it does that too).

Aren't these the key features for why we invented HTTP/2?

Admittedly, SSH is missing some pieces. It's lacking a notion of virtual hosts, or being able to serve different endpoints on different hostnames from a single IP address.

On the other hand, SSH *does* have several cool features over HTTP/2 though, like built-in client authentication, which removes the need for registration and remembering extra passwords.

More factoids to fill your stockings

- SSH server and client specification is fairly symmetric. Per the protocol, most of what the client can ask of a server, a server *could* ask of the client. This includes things like run commands, but mainstream clients don't implement this (as is recommended against in the specification).
- Every keystroke is sent over the TCP connection. This is why you might notice lag in your typing.
- To see what your OpenSSH client is doing, use -v to enable verbose debugging output. Use another -v to see per-keystroke debugging, and another -v to further increase the silliness.
- There is a protocol called MOSH which uses SSH to bootstrap but uses client-side predictive rendering and a UDP state synchronization protocol to remove the effects of latency. I wish there were more third-party implementations of it.
- Since SSH supports port forwarding and a SOCKS proxy interface, you can build a VPN on top of it by using something like sshuttle.
- SSH can authenticate using a certificate authority scheme, similar to x.509 certificates used in TLS. Also, many clients can verify server fingerprints against an *SSHFP* DNS entry.

Some provocative SSH ideas

Chat over SSH was fun, but that's just the tip of what we can come up with.

Multi user dungeon (MUD)

Someday, you'll be able to ssh into *mud.shazow.net* and you'll get a little ASCII RPG world to explore. Not yet, but it just might happen.

Distributed hash table (DHT)

This gets technical but the possibilities are striking…https://twitter.com/shazow/status/549348566972370944

Programmatic data streams

Or better yet, ZeroMQ-style sockets with proper security and encryption? Check out Jeff Lindsay's Duplex. Still a proof of concept, but lots of really cool demos.

RPC API

SSH's built-in authentication and encryption makes it *really* convenient for things like APIs. No complicated OAuth2 handshakes or HMACs and signatures.

```
ssh api.example.com multiply a=4 b=5
```

Someday we'll have good libraries which make connecting over SSH just as easy as HTTP. At that point, your code will look exactly like today's run-of-the-mill REST API, but use SSH underneath.

Either way, the days of *curl* examples in API docs would be behind us.

File server

If we have an RPC API, why not serve static files while we're at it?

```
ssh static.example.com get /images/header.png
```

Remember, SSH supports persistent connections just as well, so your browser could sit there connected to an SSH channel named *get* for the host and send concurrent *get* requests for assets. We could even implement ETAGs, and whatever else.

And finally, HTTP

At this point, there's no reason we couldn't build a version of HTTP/1 or HTTP/2 on top of SSH. Let's add a *header* channel to specify things like *Host* for virtual host support, throw in some *Cookie* headers too. Want to add some method verbs? Why not, let's make a bunch of channels like *post* or maybe *http-post* if we want to be polite.

Why aren't we using SSH for everything?

Great question. ■

Software design patterns are not goals, they are tools

By MATTHEW JONES

I went through a phase earlier in my career where I thought design patterns were the be-all, end-all of software design. Any system which I needed to design started with the applicable patterns: Factories, Repositories, Singletons, you name it. Invariably, though, these systems were difficult to maintain and more difficult to explain to my coworkers, and I never quite seemed to put two and two together. The fact was that I just didn't understand them the way I thought I did.

Five years later, I'm now researching these same design patterns for a presentation I'm giving at my day job, the goal of which is to demonstrate how said patterns help us maintain our projects in a C#/.NET environment. I've built and documented several examples for Adapter, Facade, Abstract Factory, etc. They are making sense to me, I definitely understand them more thoroughly, but there's something that still seems a little...off.

To be clear, I've never read the Gang of Four book these patterns are defined in, so it's possible there's reasoning in the book that would alleviate my concerns. In fact, all of my knowledge about these patterns has come from online resources such as Do Factory. And yet the more I understand them, the more I believe that design patterns are not goals that we should strive for.

Take the Adapter pattern as an example. Adapter strives to provide an abstraction over an interface such that a client which expects a *different* interface can still access the old one. Imagine that we have a legacy API, which looks something like this (taken directly from Do Factory above):

```
class Compound
{
    protected string _chemical;
    protected float _boilingPoint;
    protected float _meltingPoint;
    protected double _molecularWeight;
    protected string _molecularFormula;

    // Constructor
    public Compound(string chemical)
    {
        this._chemical = chemical;
    }

    public virtual void Display()
    {
        Console.WriteLine("\nCompound: {0} ------ ", _chemical);
    }
}

class ChemicalDatabank
{
    // The databank 'legacy API'
    public float GetCriticalPoint(string compound, string point)
    {
        // Melting Point
        if (point == "M")
        {
            switch (compound.ToLower())
            {
                case "water": return 0.0f;
                case "benzene": return 5.5f;
                case "ethanol": return -114.1f;
                default: return 0f;
            }
        }
        // Boiling Point
        else
        {
            switch (compound.ToLower())
            {
                case "water": return 100.0f;
                case "benzene": return 80.1f;
                case "ethanol": return 78.3f;
                default: return 0f;
            }
        }
    }

    public string GetMolecularStructure(string compound)
    {
        switch (compound.ToLower())
        {
            case "water": return "H20";
            case "benzene": return "C6H6";
            case "ethanol": return "C2H5OH";
            default: return "";
        }
    }

    public double GetMolecularWeight(string compound)
    {
        switch (compound.ToLower())
        {
            case "water": return 18.015;
            case "benzene": return 78.1134;
            case "ethanol": return 46.0688;
            default: return 0d;
        }
    }
}
```

However, our new system expects Critical Point, Molecular Weight and Molecular Structure to be properties of an object called `RichCompound`, rather than queries to an API, so the Adapter patterns says we should do this:

```
class RichCompound : Compound
{
    private ChemicalDatabank _bank;

    // Constructor
    public RichCompound(string name)
        : base(name)
    {
    }

    public override void Display()
    {
        // The Adaptee
        _bank = new ChemicalDatabank();

        _boilingPoint = _bank.GetCriticalPoint(_chemical, "B");
        _meltingPoint = _bank.GetCriticalPoint(_chemical, "M");
        _molecularWeight = _bank.GetMolecularWeight(_chemical);
        _molecularFormula = _bank.GetMolecularStructure(_chemical);

        base.Display();
        Console.WriteLine(" Formula: {0}", _molecularFormula);
        Console.WriteLine(" Weight : {0}", _molecularWeight);
        Console.WriteLine(" Melting Pt: {0}", _meltingPoint);
        Console.WriteLine(" Boiling Pt: {0}", _boilingPoint);
    }
}
```

The `RichCompound` class is therefore an adapter for the legacy API: it converts what were service calls into properties. That way our new system can use the class it expects and the legacy API doesn't need to go through a rewrite.

Here's the problem I have with design patterns like these: they seem to be something that should occur organically rather than intentionally. We shouldn't directly target having any of these patterns in our code, but we should know what they are so that if we accidentally create one, we can better describe it to others.

In other words, *if you ever find yourself thinking, "I know, I'll use a design pattern" before writing any code, you're doing it wrong*.

What patterns don't help with is the initial design of a system. In this phase, the only thing you should be worried about is how to faithfully and correctly implement the business rules and procedures.

Following that, you can create a "correct" architecture, for whatever correct means to you, your business, your clients, and your code standards. Patterns don't help during this phase because they artificially restrict what you think your code can do. If you start seeing Adapters everywhere, it becomes much harder to think of a structure that may not have a name but would fit better in your app's architecture.

I see design patterns as tools for refactoring and communication. By learning what they are and what they are used for, we can more quickly refactor troublesome projects and more thoroughly understand unfamiliar ones.

If we see something that we can recognize as, say, the Composite pattern, we can then look for the individual pieces of the pattern (the tree structure, the leaves and branches, the common class between them) and more rapidly learn why this structure was used. *Recognizing patterns help you uncover why the code was structured in a certain way.*

Of course, this strategy can backfire if the patterns were applied inappropriately in the first place. If someone used a Singleton where it didn't make sense, we might be stuck, having assumed that they knew what they were doing.

Hence, don't use patterns when beginning your design of a project; use them after you've got a comprehensive structure in place, and only where they make sense. Shoehorning patterns into places where they don't make sense is a recipe for unmaintainable projects.

Software design patterns are tools, not goals. Use them as such, and only when you actually need them, not before. ■

How Tor works

By ALEXANDER KYTE

Introduction

The Tor network is a public distributed network that uses cryptography to provide probabilistic anonymity. It establishes TCP streams from one end of a global cluster of untrusted proxies to the other, hiding traffic source information while allowing bidirectional network traffic over the stream. We will study how intelligent network architecture as well as cryptography allows for this real-life system to provide service in the face of coordinated attacks by very powerful adversaries.

Threat model and goals

Tor differs from many principled distributed architectures in that the primary concern is usability. Rather than making promises about the security of the system and accepting usability restrictions, Tor promises that it can be used for Web traffic and interactive applications. As we will see later, this informs the decision to accept a certain level of vulnerability to timing attacks.

At first, Tor's goal was to allow for privacy of traffic. It should be unfeasible for an eavesdropper or hostile network member to uniquely identify the source of a given TCP packet that leaves the network. As many people who require privacy live in countries where the network is blocked, state-level attackers comprise many of Tor's serious adversaries. This expanded Tor's threat model to include resistance to censorship from an adversary capable of enumerating many machines and completely controlling traffic within an IP range. This has motivated changes in relay discovery since the initial Tor architecture.

Onion routing 101

Imagine that you wanted to get a packet from Alice's computer to Bob's web server. Let's say that Alice lives in a country where people who shop for pressure cookers and backpacks get raids from federal agents. Now Bob sells a lot of household items and unfortunately his ISP has been compelled by the government to give them all of his traffic. Alice doesn't want to get raided because her neighbors are light sleepers. How can we get Alice's packet to Bob without giving the government somebody to intimidate, pay, or torture to get the information?

So the first idea is for Alice to have a friend carry the packets along for her. She sets up a TLS connection with a web proxy on her friend's server. This friend lets anybody on the Internet use it, so the government has no idea which of the 10k citizens are shopping for things on the naughty list. This has some terrible properties though.

This server is now "trusted." This isn't "trusted" in the sense that it's trustworthy, but "trusted" in that you must trust this relay with the entirety of your security. If this server is run in Alice's country, or her country is an ally of the country the server runs in, her friend running the server might get raided. Friends don't get friends raided. Also, the issue of timing and packet size comes into play. If an attacker watches all of the traffic coming in and out of the proxy, and if Alice is the only one sending a packet every 50ms with a 2kb payload, then the attacker can trivially associate Alice's input with the proxy's output.

What if you were to chain together multiple proxies? You have the same problems without cryptography. Every proxy is potentially the weakest link in the chain. You have a weaker, not a stronger system.

Now let's give every proxy a public/private keypair. You could encrypt the message for the last proxy, and now all of the other proxies can't read it. Furthermore, the last proxy only knows the address of the server sending the message to it. If the attacker shows up at their door, they can explain that they're unable to help without lying. The issue now is that all of the proxies before the last in the chain know the address of the last proxy. This means that the first proxy in the chain sees both Alice's IP address and the IP address of the last proxy; the first proxy can tell the attacker everything it needs to know.

What you do is that you encrypt the message with the public key for the last proxy first. Then you create a message for the next-to-last server saying "send this encrypted message to the proxy with this

IP". You encrypt this message with the public key for the next to last proxy. You do this for each step in the chain until you reach the first proxy. You can encrypt with the public key for this relay, but Tor doesn't because Tor connects Alice to the first proxy with TLS. TLS is already secured using public key cryptography.

Now we've discussed that public key cryptography is slow. If you were to use it for all of the packets as we described, you'd have a very slow network. Also, a government would be able to go to each proxy server owner and make them decrypt the given packet because the proxy's public/private keypair remains the same.

Instead, we can use public key cryptography to agree on a symmetric encryption key for each server. Now we still encrypt the packet in layers from last relay to first relay, but we use a temporary key that is fast to use. At the end of every hour or so, all of these proxies must throw away their keys. It's typically outside of an attacker's power to seize many servers spanning many countries within an hour. This is called perfect forward secrecy because a security breach in the future doesn't jeopardize the security of past traffic.

This doesn't prevent the timing attack which we spoke about. The Tor network could work around this problem by collecting all of the messages to send in each 10-minute period, add padding on the end of each message to get a uniform size, and shuffle them all around. The Tor project chose not to do this because it would make the system unsuitable for interactive web browsing. Instead, the Tor network has decided that this is a failure state. The Tor project has a number of strategies to make it unlikely for a single attacker to be able to observe enough of the exit relays and enough of the entry relays into the network to correlate messages via size and timing. These don't always work. Later we will cover the most recent and most successful attack on the Tor network.

Circuits and connection pooling

If you need to service many requests, it's very important to balance the network load across many machines. When a client needs to use Tor, they will create a route through the network with an entry point and an exit point. This is called a circuit.

Onion routing as it had existed before Tor would simply create a new circuit for each TCP connection that the user needed. The issue with this is that some web traffic patterns can create and dispose of hundreds of connections regularly. In order to achieve the promise of usability, Tor has an interesting connection pooling architecture that tries to reuse routes through the network (called circuits).

When a client tries to connect to the network, they will first get a list of the relays they can use. From this, they will choose an exit relay, a guard (entry) relay, and one intermediate relay. Tor will make sure that these relays are geographically far apart, and are not hosted by the same people (self-reported).

We will later explain the differences in relay types. We only choose 3 hops because adding more hops doesn't really add security. Each relay can only know the predecessor and successor, so 3 hops will provide the required amount of isolation. Furthermore, circuits with more hops must go through more servers, increasing the likelihood that an attacker's server is included in this chain.

Once a client has chosen the relays to use, it will connect to the guard relay over TLS. Since TLS uses secure, fast communication with good key negotiation, there is no other crypto used between the client and the guard server. The client will then send "Cells" (Tor protocol packets), which tell the guard relay to connect to the intermediate relay. Cells can be commands for relays, or can be data payload packets. Further cells are sent to the intermediate relay in order to carry out a Diffie-Hellman key negotiation. After this is done, the intermediate relay is likewise commanded to extend the circuit to the exit relay.

It's worth noting that it is important to use integrity checking on these data streams. While an attacker can't read the messages, if an attacker could guess the plaintext then they could find out the stream of bits that were used to encrypt this specific packet. Future packets are not compromised by this, but the attacker gains the ability to modify the message. Certain steps in the protocol can be modified, and the network can have protocol invariants broken.

TCP packets that are sent through Tor after this circuit is built will move through these connections, encrypted symmetrically for each relay. At the exit relay, the final plaintext message is visible. Since circuits are shared for different TCP connections, it is the duty of the exit relay to make sure that the packet is put into the correct TCP stream. This works in most cases, but puts the burden of traffic cleanliness on the user. A classic example is the issue of BitTorrent.

BitTorrent will embed the IP address of the user in the data packets. This means that anybody seeing a BitTorrent packet leave the Tor network can associate it with the client on the other end. Even worse, an attacker can associate this exit relay with the circuit that is tunneling all of the client's TCP streams.

Because of attacks like this, Tor now uses a new circuit for certain kinds of traffic, such as building a circuit, sending a message, hosting a hidden service, or joining as a relay or exit relay. BitTorrent is still bad to run over Tor, but most Tor clients can try to prevent it from sharing circuits with Web traffic.

Directory system

Alice has a messy problem in the above system. If she doesn't know any of the proxy operators, how does she find a collection of trustworthy proxies to route in this onion method? If the relays that she chooses is from a small enough collection, then the risk of identification increases. How does she prevent an attacker from sending her a list of relays that only include bad relays?

This is where Tor's directory system comes into play. The directory system is a collection of router descriptors and states. The directory system of Tor is actually not distributed. It's decentralized to a degree. There are currently eight directory authorities for the deployed Tor cluster, but it managed to operate with only three for the longest time. These authorities have their public keys hard-coded into the software distribution of Tor. That this reliance upon a system of a handful of servers has remained cryptographically protected against nation-states for so long is quite a surprise.

This system has gone through three changes. At first, these servers were fully trusted. Clients would connect to all three servers and would get the state of the cluster from the most recent one spoken to. They got this document directly and it was sent without encryption.

When authorities disagreed, clients would get different documents. Even worse, an attacker could tell which document each client got. They knew which relays clients could use. Also, there was no consensus between all three servers that was required.

Lastly, each server was subjected to the load of the entire cluster. This fundamentally threatened the ability of Tor to scale.

1. The first major change was to separate the data into a document of relay attributes and a document of relay states. The latter could be used to keep a consistent view of the subset of the cluster that is available both during the state fetch and the last descriptor fetch. This state document was much cheaper to send.
2. The second change was to prevent rogue authorities from controlling a client's view of the network. This was done by fetching a more complex document from each authority that represented their view of the network. This allowed clients to find the intersection of trust between the authorities, potentially partitioning clients based on document variation throughout time. Since the documents were sent unencrypted, fingerprinting was feasible. Attacks in this vein could allow an attacker to eventually narrow down traffic sources. This moved more load from the authorities to the clients. Connecting to the network now required fetching a few megabytes of descriptors.
3. The last major change was to explicitly create a consensus document. The authorities will find a consensus view of the network once per hour using an ad-hoc consensus algorithm. They will all sign this view of the network and distribute it for the next hour. This document used a much compressed format of router descriptors, saving space. This gives all clients a consistent view of the network and decreases the footprint of the directory documents. Clients will now sometimes create a one-hop Tor connection to fetch the next document. This prevents an attacker from fingerprinting documents and profiling which clients see which network states propagate.

How does a relay join the Tor cluster?

What's interesting about the Tor cluster is that bandwidth allocation is integrated into it. A non-exit relay starts trying to join the cluster by forming four circuits to itself and sending a benchmark self-test to measure the bandwidth it can carry. This is put into a router descriptor and signed, and is offered to the directory authorities. Here we see that the directory authorities are acting as public key distribution agents much like CA authorities do in the HTTP world.

At first, the cluster doesn't trust you. It won't really route much traffic (if any) through you. This routing is done by reporting your bandwidth availability in the consensus document. Clients will choose you with a probability proportional to your bandwidth availability, and so few clients will choose you for a circuit. All the while though, relays in the network will measure you. If they find you fast and stable, they'll slowly allocate more and more traffic to you. As your traffic slice increases, your circuit speeds will be measured to find out what you can really provide.

You'll get more traffic but it will be limited still because you won't be a guard relay. If every circuit is new then the chance of not picking an attacker's relays for the first and last hops goes to 0 over enough time. If you remain with the first hop over time, then you either choose an attacker and lose any time your other relays are attacker relays, or you choose a good relay and all of your paths are safe. This stable first relay must be reliable or the performance will be terrible. This relay is a guard relay. This is essentially a way to make it more probable to be safe.

For you to become a guard relay you must be stable for 8 days. After you're a guard relay you'll be moved up into a separate class of relay, which is rotated through by clients over time. Eventually, the number of clients that cycle out of you will equal the number that cycle in. You will have your bandwidth mostly saturated at this point.

If you were an exit relay, you would see traffic leaving you with speed once your bandwidth allocation was being scaled up.

Conclusion

We see here that the use of cryptography as well as the properties of network bandwidth and geographic IP allocation allows the Tor network to put up a strong defense against adversaries with nation-state resources.

It's interesting to see here the architectural assumptions that allowing some relays to be evil requires. One must protect against data traffic injection, misuse of the protocol, observation of traffic, malicious saturation of the network, and malicious dishonesty about traffic statistics. And it's through the miracle of public key cryptography that we can negotiate data transmission contracts with potentially hostile relays.

I find the heterogeneous nature of Tor quite interesting. There is the decentralized cluster state authority system that uses a small number of trusted people to verify the truth of network observations. The actual routing policy partitions relays into where they will be on the paths through the relays. In each and every step, a relay is only allowed to perform the steps that we expect it to achieve from past behavior. Throughout all of this, we see that a fungible resource is used to make attacking Tor from the inside expensive. This resource is bandwidth over time; this resource is identity.

Attackers tend to use anonymity to make attacks cheap to carry out. A computer can participate in a denial of service attack or inject traffic frequently by changing IP address. We do not put a relay in a position to degrade the network until it has behaved well for a long time. After placing it there, we constantly observe it. When it behaves badly, we eject it from the network. We see that the cost of attacking Tor is the maintenance and loss of a cryptographically-verified trustworthy identity, which must be maintained at the cost of network bandwidth. By assuming that every network member is anonymous, the Tor protocol intelligently creates an economy of trust and makes untrustworthiness expensive. ■

Resources

- http://css.csail.mit.edu/6.858/2014/readings/tor-design.pdf
- https://www.youtube.com/watch?v=rIf_VZQr-dw
- https://blog.torproject.org/blog/top-changes-tor-2004-design-paper-part-1
- https://blog.torproject.org/blog/top-changes-tor-2004-design-paper-part-2
- https://blog.torproject.org/blog/top-changes-tor-2004-design-paper-part-3
- https://blog.torproject.org/blog/lifecycle-of-a-new-relay
- https://www.torproject.org/docs/hidden-services.html.en
- http://arstechnica.co.uk/security/2016/08/building-a-new-tor-that-withstands-next-generation-state-surveillance/

Destroy all Ifs — a perspective from functional programming

By JOHN DE GOES

The Anti-IF Campaign currently stands at 4,136 signatures, and there's a reason: conditional statements are frequently a troublesome source of bugs and brittle logic, and they make reasoning about code difficult because they multiply the code paths.

The problem is not necessarily with conditionals: it's with the boolean values that are required to use conditionals. A boolean value reduces a huge amount of information to a single bit (`0` or `1`). Then on the basis of that bit, the program makes a decision to take one path or a totally different one.

What could possibly go wrong — right?

The root problem

I've long argued that, despite all its flaws, one of the great things about functional programming is its intrinsic *inversion of control*.

In an impure imperative language, if you call some function `void doX(State *state)`, the function can do anything it wants. It can modify state (both passed in and global), it can delete files, it can read from the network, and it can even launch nukes!

In a pure functional language, however, if you call some function `doX :: State -> IO ()`, then at most it's going to return a value. It can't modify what you pass it, and if you like, you can ignore the value returned by the function, in which case calling the function has no effect (aside from sucking up a little CPU and RAM).

Now granted, `IO` in Haskell is *not* a strong example of an abstraction that has superior reasoning properties, but the fundamental principle is sound: in pure functional programming languages, the control is *inverted* to the caller.

Thus, as you're looking at the code, you have a clearer sense of what the functions you are calling can do, without having to unearth their foundations.

I think this generalizes to the following principle:

Code that inverts control to the caller is generally easier to understand.

(I actually think this is a specialization of an even *deeper* principle — that code which destroys information is generally harder to reason about than code which does not.)

Viewed from this angle, you can see that if we embed conditionals deep into functions, and we call those functions, we have lost a certain amount of control: we feed in inputs, but they are combined in arbitrary and unknown ways to arrive at a decision (which code path to take) that has humongous ramifications on how the program behaves.

It's no wonder that conditionals (and with them, booleans) are so widely despised!

A closer look

In an object-oriented programming language, it's generally considered a good practice to replace conditionals with polymorphism.

In a similar fashion, in functional programming, it's often considered good practice to replace boolean values with algebraic data types.

For example, let's say we have the following function that checks to see if some target string matches a pattern:

```
match :: String -> Boolean -> Boolean -> String -> Bool
match pattern ignoreCase globalMatch target = ...
```

(Let's ignore the boolean return value and focus on the two boolean parameters.)

Looking at the `match` function, you probably see how its very signature is going to cause lots of bugs. Developers (including the one who wrote the function!) will confuse the order of parameters and forget the meaning.

One way to make the interface harder to misuse is to replace the boolean parameters with custom algebraic data types:

```
data Case = CaseInsensitive | CaseSensitive
data Predicate = Contains | Equals
match :: String -> Case -> Predicate -> String -> Bool
match pattern cse pred target = ...
```

By giving each parameter a type, and by using names, we force the developers who call the function to deal with the type and semantic differences between the second and third parameters.

This is a big improvement, to be sure, but let's not kid ourselves: both `Case` and `Predicate` are still "boolean" values that have 1 bit of information each, and inside of `match`, *big decisions* will be made based on those bits!

We've cut out *some* potential for error, but we haven't gone as far as our object-oriented kin, because our code still contains conditionals.

Fortunately, there's a simple technique you can use to purge almost all conditionals from your code base. I call it, *replace conditional with lambda*.

Replace conditional with lambda

When you're tempted to write a conditional based on boolean values that are passed into your function, I encourage you to rip out those values, and replace them with a lambda that performs the *effect* of the conditional.

For example, in our preceding `match` function, somewhere inside there's probably a conditional that looks like this:

```
let (pattern', target') = case cse of
  CaseInsensitive -> (toUpperCase pattern, toUpperCase target)
  CaseSensitive   -> (pattern, target)
```

This normalizes the case of the strings based on the sensitivity flag.

Instead of making a decision based on a bit, we can pull out the logic into a lambda that's passed into the function:

```
type Case = String -> String
data Predicate = Contains | Equals
match :: String -> Case -> Predicate -> String -> Bool
match pattern cse pred target = ...
  let (pattern', target') = (cse pattern, cse target)
```

In this case, because we are accepting a user-defined lambda, which we are then applying to user-defined functions, we can actually perform a further refactoring to create a match combinator:

```
type Matcher = String -> Predicate -> String -> Bool
caseInsensitive :: Matcher -> Matcher
match :: Matcher
```

Now a user can choose between a case sensitive match with the following code:

```
match a b c                 -- case sensitive
caseInsensitive match a b c -- case insensitive
```

Of course, we still have another bit and another conditional: the predicate flag. Somewhere inside the `match` function, there's a conditional that looks at `Predicate`:

```
case pred of
  Contains -> contains pattern' target'
  Equals   -> eq pattern' target'
```

This can be extracted into another lambda:

```
match :: String -> (String -> Bool) -> String -> Bool
```

Now you can test strings like so:

```
caseInsensitive match "foo" eq       "foobar" -- false
caseInsensitive match "fOO" contains "foobar" -- true
```

Of course, now the function `match` has been simplified so much, it no longer needs to exist, but in general that won't happen during your refactoring.

Note that as we performed this refactoring, the function became more general-purpose. I consider that a side-benefit of the technique, which replaces highly-specialized code with more generic code.

Now, let's go through a real world case instead of a made-up toy example.

psc-publish

The PureScript compiler has a publish tool, with a function defined approximately as follows:

```
publish :: Bool -> IO ()
publish isDryRun =
  if isDryRun
    then do
      _ <- unsafePreparePackage dryRunOptions
      putStrLn "Dry run completed, no errors."
    else do
      pkg <- unsafePreparePackage defaultPublishOptions
      putStrLn (A.encode pkg)
```

The purpose of `publish` is either to do a dry-run publish of a package (so the user can make sure it works), or to publish the package for real.

Currently, the function accepts a boolean parameter, which indicates whether or not it's a dry-run. The function branches off this bit to decide the behavior of the program.

Let's unify the two code branches by extracting out the options, and introducing a lambda to handle the different messages printed after package preparation:

```
type Announcer = String -> IO String

dryRun :: Announcer
dryRun = const (putStrLn "Dry run completed, no errors.")

forReal :: Announcer
forReal = putStrLn <<(A.encode pkg)

publish :: PublishOptions -> Announcer -> IO ()
publish options announce = unsafePreparePackage options >>= announce
```

This is better, but it's still not great because we have to feed two parameters into the function `publish`, and we've introduced new options that don't make sense (publish with dry run options, but announce with `forReal`).

To solve this, we'll extend `PublishOptions` with an `announcer` field, which lets us collapse the code to the following:

```
forRealOptions :: PublishOptions
forRealOptions = ...

dryRunOptions :: PublishOptions
dryRunOptions = ...

publish :: PublishOptions -> IO ()
publish options = unsafePreparePackage options >>= announcer options
```

Now a user can call the function like so:

```
publish forRealOptions
publish dryRunOptions
```

There are no conditionals and no booleans, just functions. You never need to decide what bits mean, and you can't get them wrong.

Now that the control has been inverted, the *caller* of the `publish` function has the ultimate power, and can decide what happens deep in the program merely by passing the right functions in.

Summary

If you don't have any trouble reasoning about code that overflows with booleans and conditionals, then more power to you!

For the rest of us, however, conditionals complicate reasoning. Fortunately, just like OOP has a technique for getting rid of conditionals, we functional programmers have an alternate technique that's just as powerful.

By replacing conditionals with lambdas, we can invert control and make our code both easier to reason about and more generic.

So what are you waiting for?

Go sign the Anti-IF Campaign today. :)

P.S. I'm just joking about the Anti-IF campaign (but it's kinda funny).

Addendum

After writing this post, it has occurred to me the problem is larger than just boolean values.

The problem is fundamentally a *protocol* problem: booleans (and other types) are often used to encode program semantics.

Therefore, in a sense, a boolean is a serialization protocol for communicating intention from the caller site to the callee site.

At the caller site, we serialize our intention into a bit (or other data value), pass it to the function, and then the function deserializes the value into an intention (a chunk of code).

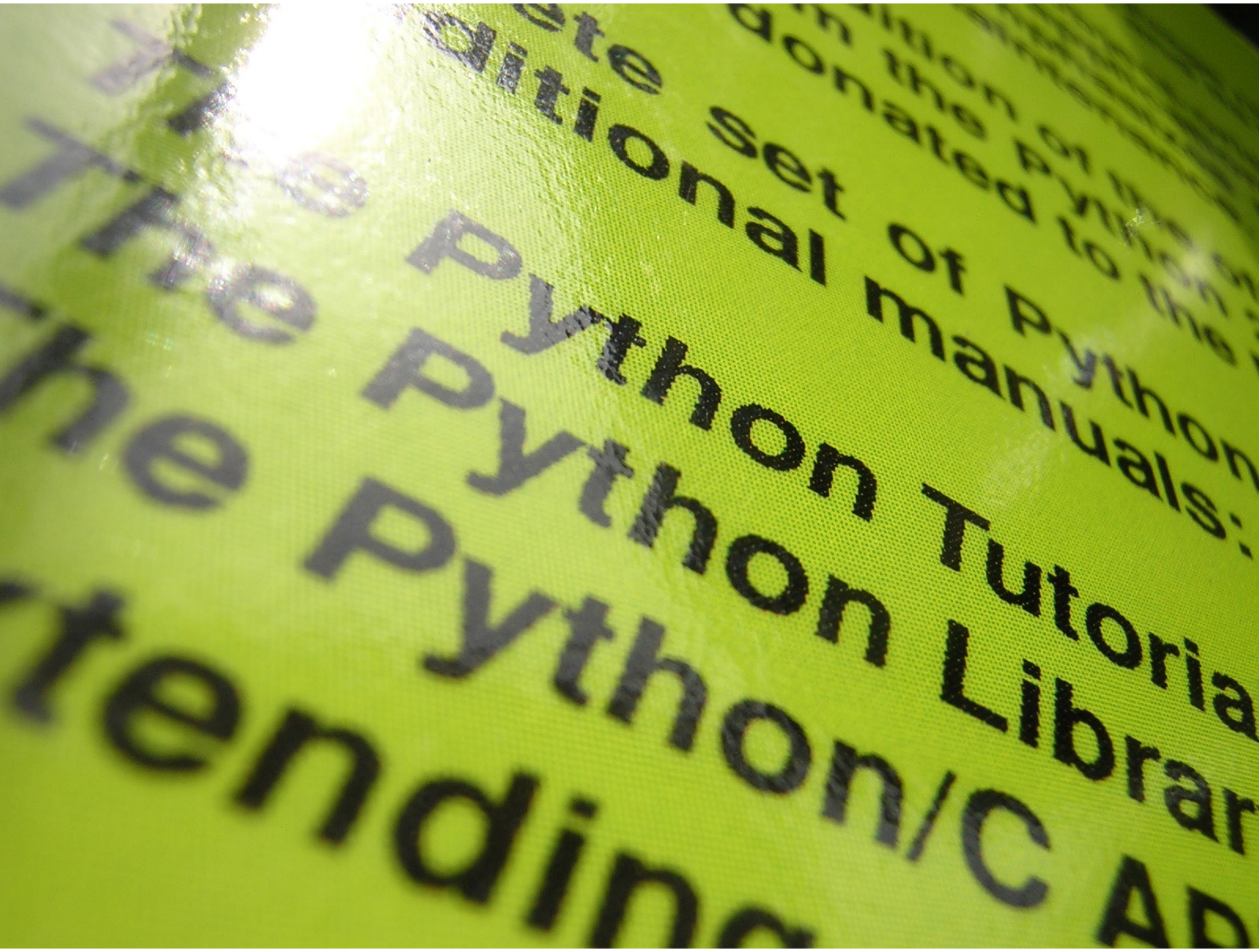
Boolean blindness errors are really just *protocol errors*. Someone screwed up on the serialization or deserialization side of things; or maybe the caller or callee misunderstood the protocol and thought different bits had different meanings.

In functional programming, the use of lambdas allows us to propagate not merely a *serialized* version of our intentions, but our *actual* intentions! In other words, lambdas let us pull the intentions out of the callee site, and into the caller site, since we can propagate them directly without serialization.

In older programming languages, this could not be done: there was no way to ship semantics (a hunk of code) from one part of the program to another. Today's programming languages support this style of programming (to varying degrees), and functional programming languages encourage it.

Inversion of control, in this limited sense, is about removing the possibility of a protocol error by giving the caller direct control over semantic knobs exposed by the callee.

Kinda cool, eh? ■



"From the light came Python 3K" by Francois Schnell is licensed under CC BY 2.0

Why you should learn Python

By ILLYA GERASYMCHUK

Introduction

My first encounter with Python was a part of the introductory course to programming. Well, I actually played with it on my own before, so I was already familiar with its syntax when the course began, but I didn't do any real project in it before that course. Even though I thought it's a great language to introduce people to programming, I wasn't a big fan of it. It's not that I disliked the language, it was more of a "meh" attitude. The reason was simple: there was "too much magic". Coming from a background of languages such as C and Java, which are a lot more explicit in terms of what's going on under the hood, Python was the complete opposite of that.

Another issue was that Python seemed a lot less structured: writing large, complex programs seemed to be a tougher task to achieve than, for example in Java, where you have some strict rules when it comes to the structure of the program (for instance the one public class per file rule), Python on the other hand, gives you a lot more freedom in such things.

Another thing is strict typing and debugging: since Python is an interpreted language, finding bugs wasn't as easy: if you have a syntax error in C, the program will simply not compile, on the other hand, in interpreted languages, the problem might go unnoticed for quite some time, until the execution reaches that particular line of code.

Trying to pass a string where an integer is expected? `cc` will go crazy at you, while Python's interpreter won't mind at all (there are some tool that address that problem though, like mypy, but I'm talking about vanilla Python). What I just mentioned here is a general downside of interpreted languages and is not exclusive or particular to Python, but those were some of the main reasons of my initial attitude towards it.

One more thing that I found a little annoying is the *required indentation*. Our teachers (who were great, by the way!) sold that as being a "good thing", since "it forced us to a cleaner code writing style". And that is true, but it was a bit annoying when your code doesn't work as expected, and you analyze the code trying to hunt the bug but can't seem to find it, until after some time you notice that one of the lines in your `if` statement has an extra space.

I had a discussion with a colleague about Python, telling him how I'm not sure why I wasn't a huge fan of the language before, and he asked me in a laughing tone "What's not to like about Python? The fact that it reads almost like English?" The answer to that question is "yes". Since the language does so many things for you under the hood, sometimes it's not clear what's happening. Let's take as an example file reading. Suppose you want to read the contents of a file and print them out, line by line. Here's how you could do it in C:

```
#include <stdio>

int main(void) {
    FILE *fp;
    char buff[256]; // assuming a line won't contain more than 256 chars
    fp = fopen("hello.txt", "r");

    while(fgets(buff, 256, fp)) {
        printf("%s", buff);
    }

    fclose(fp);
    return 0;
}
```

Now the same thing in Python:

```
with open('hello.txt') as f:
    for line in f:
        print(line)
```

Now, many people will consider that as an advantage, however, while in the first case it's pretty clear what's happening:

1. We obtain a file pointer to a file
2. Read the bytes from each line into a buffer, and then print that line from that same buffer
3. Close the stream

In the Python's example none of that is obvious, it just sort of "magically" works. Now, while one might argue that it's a good thing, since it abstracts the programmer away from the implementation details (and I agree with that), I like to know exactly what's happening.

It's interesting that many of the things that I mentioned as disadvantages, I now consider advantages. To be fair, there is no "magic" in Python — if you dive in a little deeper, you'll find out that there is no actual magic involved, it's just the way the language interprets your code, and from that perspective, I find it fascinating. If you share the same feelings, I suggest you investigate further about how the language works — if something seems like "magic", find out what's actually happening, and things will become a lot clearer, and that "magic" will turn into "convenience".

My opinion on those points has changed a lot, especially after I decided to give the language another go. In fact, I'm now a big fan of Python! Now you might be wondering when I'm going to try to convince you that learning Python is a good idea. Don't worry, that part is coming next. As a closing point to the introduction, I want to mention that this is my personal feeling towards the language, and it is just a *personal preference*. I didn't try to convince people that they should learn C, because if you're programming in Python "you're not a real programmer" (in fact, I don't believe in that).

When people asked me which language they should learn as their first, I usually suggested Python, for many of the reasons that I mentioned above as "disadvantages". My feelings were mostly based on my personal interests, I was doing some more low-level stuff at the time, so as you might imagine, Python didn't fit in that picture.

Python: the good parts

After jacking the title for this section from a popular (and great) JavaScript book, it's time to begin the topic of this blog post: why you (yes, you!) should learn Python.

1. Universal scripting language

This was one of the main reasons why I decided to give Python a second go. I was working on various projects with various people, and naturally different people used different operating systems. In my case, I was usually switching between Windows and Linux.

To give a concrete example, on one of the projects I wrote a script that automated testing in a project, only to realize that I was the only one who was taking advantage of it, since it was written in `PowerShell` and I was the only one using Windows. Now there was natural "bash is so much better" from my colleagues, while I tried to explain to them that `PowerShell` followed a completely different paradigm and that it had its strong points (for example, it exposes the `.NET` framework interface), and that it's an Object-Oriented scripting language, quite different from `bash`. Now I'm not going to discuss which one is better, since that's not the focus of this post.

So how could this problem be solved? Hmmm… now, is there a language that is script-friendly and runs on all major operating systems? You've guessed it, that language is `Python`. Besides running on all major operating systems, it also includes functionality useful for scripting out of the box. The standard library includes a handful of utilities that provide a common interface for operating system dependent functionality.

To provide a simple and straightforward example, let's assume that you wanted to get a list of names of all files in a directory and then do something with those names. In UNIX, you'd have something like:

```
for f in *; do echo "Processing $f file..."; done
```

While in PowerShell, you'd go with something similar to:

```
Get-ChildItem "." |
Foreach-Object {
    $name = $_.Name
    Write-Output "Processing $($name) file..."
}
```

An equivalent functionality in Python can be achieved with:

```
from os import listdir

for f in listdir('.'):
    print('Processing {} file...'.format(f))
```

Now, besides running on Linux, MacOSX and Windows, in my opinion, it's also more readable. The example above is a very simple script, for more complex examples the difference in readability is even more obvious.

As I mentioned earlier, Python comes with great libraries out of the box, for the purpose of replacing shell scripting, the ones that you'll find the most useful are:

- os — provides OS independent functionality, like working with paths and reading/writing files.
- subprocess — to spawn new processes and interact with their input/output streams and return codes. You can use this to launch programs already installed on your system, but note that this might not be the best option if you're worried about the portablity of your script.
- shutil — offers high-level operations on files and collections of files.
- argparse — parsing command-line arguments and building command-line interfaces

Alright, let's say you get the point, cross-platformness (is that even a real word?) and readability sounds great, but you really like the UNIX-like shell syntax. Good news, you can have the best of both worlds! Check out Plumbum, it's a Python module (more on that topic later on), whose motto is *"Never write shell scripts again"*. What it does is it mimics the shell syntax, while keeping it cross-platform.

You don't have to ditch shell scripting altogether

While it's possible to completely substitute shell scripts with Python, you don't have to, since Python scripts naturally fit into the command chaining philosophy of UNIX, all you have to do is make them read from `sys.stdin` (standard input) and write to `sys.stdout` (standard output).

Let's look at an example. Let's say that you have a file with each line containing a word and you want to know which words appear in the file and how many times. This time we don't want to go "all Python", we're actually going to chain the `cat` command with our Python script, which we'll call `namecount.py`.

Let's say we have a file called `names.txt` with the following content:

```
cat
dog
mouse
bird
cat
cat
dog
```

This is how our script will be used: `$> cat names.txt | namecount.py`. And `PowerShell` folks: `$> Get-Content names.txt | python namecount.py`.

The expected output is something like (order might vary):

```
bird    1
mouse   1
cat     3
dog     2
```

Here is the source of `namecount.py`:

```
#!/usr/bin/env python3
import sys

def count_names():
    names = {}
    for name in sys.stdin.readlines():
        name = name.strip()

        if name in names:
            names[name] += 1
        else:
            names[name] = 1

    for name, count in names.items():
        sys.stdout.write("{0}\t{1}\n".format(name, count))

if __name__ == "__main__":
    count_names()
```

Having the information displayed unordered is not the most readable thing, and you'll probably want that ordered by the number of occurrences, so let's do that. We'll use the piping mechanism again and offload the job of sorting our output to the built-in commands. To sort our list numerically, in descending order, all we have to is `$> cat names.txt | namecount.py | sort -rn`.

And if you're using `PowerShell`:`$> Get-Content names.txt | python namecount.py | Sort-Object { [int]$_.split()[-1] } -Descending` (you can almost hear the UNIX folks complain about how verbose the PowerShell version is).

This time our output is deterministic and we'll get:

```
cat     3
dog     2
bird    1
mouse   1
```

(As a side-note, if you're using `PowerShell`, `cat` is an alias for `Get-Content` and `sort` is an alias for `Sort-Object`, so the commands above can be also written as: `$> cat names.txt | python namecount.py` and `$> Get-Content names.txt | python namecount.py | sort { [int]$_.split()[-1] } -Descending`)

Hopefully I have convinced you that Python might be a good substitute for at least some of your scripts and that you don't have to ditch shell scripting altogether, since you can incorporate Python scripts into your existing workflow and toolbox, with the added benefits of it being cross-platform, more readable and having a rich library collection at your disposal (more on that later on).

2. A lot of great libraries

Python has a very rich library collection. I mean, there is a library for almost anything (fun fact: if you type `import antigravity` in your Python interpreter, it opens a new browser window, which leads you to that xkdc comic. Now how awesome is that?).

I'm not a big fan of programming just by "stacking one library onto another", but you don't have to. Just because there are a lot of libraries, it doesn't mean that you have to use them all. While I don't like to just stack libraries one onto another (which looks more like CBSE), I obviously recognize their use and do use them.

For example, I decided to play around with Markov Chains, so I came up with an idea for a project: grab all of the lyrics from an artist, build a Markov Chain with that and then generate songs from them. The idea is that the generated songs should reflect the artists' style. So I hacked around with that project for a bit, and the result was lyricst (this is more like a proof of concept than a complete, fully tested project. As I said, I just hacked around for a bit, so don't go too hard on it. It does include a command line-interface and some documentation with examples, if you want to play around).

I decided that a great place to get lyrics from would be RAPGenius, since it's actively used and usually up to date).

So to get all of the artists' lyrics, I would have to scrape them from the website and work with HTML. Luckily, Python is great for web scraping and has great libraries like BeautifulSoup to work with HTML. So this is what I did: Iused BeautifulSoup to get all of the info from the page I needed (which was basically the song lyrics) and then used that info to build MarkovChains. Of course I could've used regular expressions or built my own HTML parser, but the existence of such libraries allowed me to concentrate on what was the real goal for this project: play around with Markov Chains, while also making it more interesting than let's say, just reading some text from files.

3. Great for pentesting

If you're into penetration testing or simply like to hack around with stems, Python is your friend! The fact that Python comes pre-installed on almost any Linux and MAC OS machine, has a rich library collection, very comprehensive syntax and is a scripting language, makes it a great language for that purpose.

Another reason why I decided to give Python a second go (besides the one mentioned in the previous section) is that I'm interested in security and Python seemed like a perfect choice for pentesting. One of my first encounters in that world was Scapy (or Scapy3k, for Python3) and let me tell you, I was impressed. Scapy is used for network packet creation, capture and manipulation. It has a straightforward API and great documentation. You can easily create packets on various layers (I'm talking about the OSI model) or capture them for analysis and modification. You can even export the `.pcap` files and open them in WireShark. It doesn't stop at network packet capture though, and there is a wide array of great libraries for that purpose. But I'm not going to cover them here, since that is not the topic of this post and it deserves a post just by itself.

Some of you might say, "Oh, that's great, but I'm interested in exploiting Windows machines, and those don't come with Python pre-installed". No worries, you can always compile your Python script to a standalone `.exe`, using py2exe. The executables can get a little big (depending on the number of libraries you're using in your script), but usually it's nothing major.

If you're intrigued, however, check out this list of Python pentesting tools. At the end of this post I also include some book recommendations.

4. A hacker's language

Python is a very malleable language. You can customize the way it works in many ways. From altering the way imports work to messing with classes before they are created. Those are just some of the examples. This also makes it a very powerful scripting language (as mentioned in section 1) and is great for pentesting (section 3), since it gives you a lot of freedom with your scripts.

I won't go deep into those topics, but I will describe the "WOW" moment that I had with this. So, I was doing some webscraping (Python is great for this task!), and one of the tools I used was BeautifulSoup. This was one of my "learning Python" projects. BeautifulSoup's syntax for working with HTML is very clean and intuitive, and one of the reasons for that is the fact that Python gives you a lot of freedom when it comes to customizing its behavior. After playing a bit with the API, I noticed that there was some "magic". The situation was similar to this one:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup('<p class="someclass">Hello</p>', 'html.parser')
soup.p
```

What the code above does is it creates a BeautifulSoup instance from the string passed as the first argument. The second argument just indicates that I want to use Python's built-in HTML parser (BeautifulSoup can work with various parsers). `soup.p` returns a `Tag(bs4.element.Tag)` object, which represents the `<p>` tag passed as the first argument.

The output of the code above is:

```
<p class="someclass">Hello</p>
```

Now you might wondering, "where's the magic part you were talking about"? Well, this part comes up next. The thing is that the code above can be adapted to any tag, even the custom ones. This means that the code below works just fine:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup('<foobarfoo class="someclass">Hello</foobarfoo>', 'html.parser')
soup.foobarfoo
```

The output is the following:

```
<foobarfoo class="someclass">Hello</foobarfoo>
```

When I realized that that works just fine, I was like "whaaaaa?" Now, the first example can be easily implemented. I mean the most straightforward way is to just define an attribute (instance variable) for every possible HTML tag, and then during parsing assign values different from `None` to them, in case those elements are found.

But this explanation does not work for the second example, and there is no way that could've been done for all possible string combinations. I wanted to know what was going on and how it was working, so I cracked open BeautifulSoup's source code and started digging. It's probably no surprise that I didn't find any attributes named `p` and the parsing functions didn't assign values to them.

After some googling, I found what was going on: *magic methods*. What are *magic methods* and why are they called that? Well, informally, magic methods are methods that give the "magic" to your classes. Those methods are always surrounded by double underscores e.g. `__init__()`. They are described in the DataModel model section in Python docs.

The specific magic method that allows BeautifulSoup to have this functionality is `__getattr__(self, name)__` (`self` in Python refers the object instance, similar to `this` in Java). If we go to the docs, here's what we'll find in the first paragraph:

> *Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for self). name is the attribute name. This method should return the (computed) attribute value or raise an AttributeError exception.*

So what happens is that if you try to access an attribute that does not exist `__getattr__(self, name)` of that object will be called, with `name` being the name of the attribute you tried to access as a string.

Let me show you an example. So let's say you have a `Person` class with a `first_name` attribute. Let's give the users of our API the ability to access the same value using `name`. Here's how our class will look like:

```
class Person(object):
    def __init__(self, first_name):
        self.first_name = first_name

    def __getattr__(self, name):
        if (name == 'name'):
            return self.first_name
        raise AttributeError('Person object has no attribute \'{}\''.format(name))
```

Let's play a little with the class above in the interactive console:

```
person = Person('Jason')
>>> person.first_name
'Jason'

>>> person.name
'Jason'

>>> person.abc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __getattr__
AttributeError: Person object has no attribute 'abc'
```

This means that we can just "make up" object attributes on the fly, isn't that awesome? So you can make your `Dog` secretly be able to "meow" as well as bark:

```
class Dog(object):
    def bark(self):
        print('Ruff, ruff!')

    def __getattr__(self, name):
        if(name == 'meow'):
           return lambda:  print('Meeeeeeow')
        raise AttributeError('I don\'t know what you\'re talking about...')
```

You can also add attributes on the fly, without reflection. `object.__dict__` is a (dictionary)[https://docs.python.org/3.5/library/stdtypes.html#typesmapping] containing the the `object`'s attributes and their values (note that I said `object`.dict, `object` is an object instance, there is also a `class`.dict, which is a dictionary of the `class`es attributes).

This means that this:

```
class Dog(object):

    def __init__(self):
        self.name = 'Doggy Dogg'
```

Is equivalent to this:

```
class Dog(object):
    def __init__(self):
        self.__dict__['name'] = 'Doggy Dogg'
```

Both of the versions share the same output:

```
snoop = Dog()

>>> snoop.name
'Doggy Dogg'
```

At this point you might be thinking, this is all great, but how is this useful? The answer is simple: *magical APIs*. Have you ever used some Python library that just feels like magic? This is one of the things that allow them to be so "magical". It's not really magic though, once you understand what's happening behind the scenes.

If you're interested in that topic, check out the Description Protocol page in the docs.

The object-oriented aspect

The object-oriented aspect of Python might seem a little "hacked in". For example, there are no private instance variables or methods in classes. So if you want to make an instance variable or a method private in a class, you'll have to stick to conventions:

- Using one leading underscore (_) for non-public instance variables and methods
- Using two leading underscores (__) for instance variables and methods will mangle their name

Let's explore an example, suppose you have the following class:

```
class Foo(object):
    def __init__(self):
        self.public = 'public'
        self._private = 'public'
        self.__secret = 'secret'
```

Let's jump into the interpreter:

```
>>> foo = Foo()
>>> foo.public
'public'
>>> foo._private
'public'
>>> foo.__secret
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foo' object has no attribute '__secret'
```

As you can see, nothing is stopping you from accessing the `_private` instance variable, but what happened in the last case? Does that mean variables starting with __ are really private? The answer is no, their name is just mangled. In essence, instead of being called `__secret`, its name has been changed to `_Foo__secret` by Mr. Python himself. You can still access it, if you really want to:

```
>>> foo._Foo__secret
'secret'
```

However, PEP8 suggests to only use leading double underscores to avoid name conflicts with attributes in classes designed to be subclassed. “PEP”, stands for “Python Enhancement Proposal”, and it’s used for describing Python features or processes. If you want a new feature to be added to the language, you create a PEP, so the whole community can see it and discuss it. You can read more about PEPs here. And yes, the description of what a PEP is a PEP itself.How Meta is that?

As you can see, Python puts a lot of trust in the programmers.

I won’t go much further into the OO topic here, since again, that deserves a post (or even a series of them) just for itself.

I do want to give you a heads up that it might get some getting used to, and it might not seem as “natural” as it is in languages like Java.But you know what, it’s just a different way of doing things. As another example, you don’t have abstract classes, you have to use decorators to achieve that behavior.

Conclusion

Hopefully this post gave you some insight into why you should consider giving Python a go. This post is coming from someone who feels “guilty” for talking poorly about Python in the past and is now all over the hype train. In my defense, it was just a “personal preference thing” when people asked me about which language they should learn first, and for instance, I usually suggested Python.

If you’re still undecided, just give it a go! Try it out for an hour or two, read some more stuff about it. If you like learning from a book, I also got you covered, check out Fluent Python. The section below has some more recommendations. ■

Book recommendations

As promised, here is the book recommendation section. I will keep this one short, only listing books that I’ve read/had experience with myself.

- Fluent Python — GREAT book about Python 3,whether you are a novice, an intermediate or an experienced Python programmer. Covers the ins and outs of the language.
- Web Scraping With Python — the title says it all. It’s a book about webscraping with Python. You’ll explore how to scrape the web for content, parse HTML and much more. I would say this book is good for novice and possibly intermediate people in the webscraping area. Even if you’ve never used Python before, you can still follow along. It does not go into any advanced topics.
- Black Hat Python — oh this one’s fun! You’ll build reverse SSH shells, trojans and much more! If you want to know how Python can be used in pentesting, make sure you check this out. Please note that this book contains Python 2 code. I do have a repo, however, that uses Python 3.
- Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers — more on the same topic as above. You’ll learn how to write practical scripts for pentesting, forensic analysis and security in general.

Edits

- Thanks to David Oster, Keyaku, Chris, muellermartin and foobarwtf for pointing out some typos.
- As per jyf1987’s suggestion, here is a `namescount.py` version using `collections.defaultdict`: namescount.py using defaultdict. I’m keeping the original one in the post, because the goal was to be more explicit, so that people new to Python would have less trouble understanding it.

"Scrum Hell" by Enrique Fernández is licensed under CC BY 2.0

Why I'm not a big fan of Scrum

By ULAS TÜRKMEN

Scrum is now the default agile software development methodology. This management framework, which is "simple to understand but difficult to master", is used by 66% of all agile companies.

After two extensive workshops, more than five years, and a couple hundred of sprints working in Scrum, I have some points of criticism about it. I think it's not naturally conducive to good software, it requires too much planning effort on the part of the developers, and it inhibits real change and improvement. In the following, I will try to put these into more detail by organizing them around more concrete topics.

Before you go to the comments section to tell me that I have no idea what I'm talking about, please keep in mind a few things. First of all, this is not a rant against agile. I'm a big fan of agile, as it is explained in"The New Methodology", and I believe that the potential of this concept has not been exhausted yet.

Also, I'm not against every idea and practice in Scrum. For example, the principles of the whole team taking responsibility for the code base, or always having an integrated, working master are really awesome.

Last but not least, the following points are directed against standard Scrum as described in the official guide. If you are doing something totally different but still calling it Scrum, this post is probably not so relevant for you.

One thing I would like to refrain from here is anecdotal evidence. My individual experiences, as far as they are not related to the proper Scrum entities, are not really relevant, since many of them are individual mistakes, and will therefore intentionally be left out from the following.

Obsession with points

The use of story points appears to be one of the defining features of Scrum. Each user story is given a certain count of story points, and a team is confronted with the number of points it has "achieved" at the end of a sprint.

Some teams also follow their daily progress in terms of story points with a burndown chart that is consulted every day at stand-up. The points collected at the end of a sprint constitute the *velocity* of a team, and the team is expected to keep this velocity.

During planning, the team takes on stories it thinks it can finish until the end of the sprint by looking at the velocity from previous sprints. The velocity of the teams serves to project an estimate of what can be achieved in the future, for the purpose of business planning.

There are many murky things about story points, but somehow the scrum masters and coaches will not abandon it. First of all, what are story points? Are they measures of time it takes to complete a story? If yes, then why are they not in terms of time?

Are they measures of complexity? If yes, why are we not talking of the complexity of stories, and how we can remove them, instead of how we can achieve as many points as possible? That is, shouldn't we be talking about doing as few points as possible? The best measure I have heard is effort. You can work three hours on half effort, but work hard for an hour and finish a task, which explains why it's not about time.

No matter how you define story points, the real issue with them doesn't go away. The main purpose of points is making planning more reliable, and providing a temporal perspective for business. They never fail to take on a life of their own, however, with teams working to gather points instead of delivering good software.

I don't understand why points are special compared to the oft-mocked cases of bug hunts or lines of code written. If devs are measured on points, they will optimize on points. Has the code base improved? Did it become more modular, simpler, *habitable* (see the section *Habitability and Piecemeal Growth* in this book (pdf) by Richard P. Gabriel)? None of these questions is of relevance. The points have to be gathered. The spice has to flow. That's what counts.

One can of course counter that if you write stories for accomplishing these counter-examples, you would get points for them. But the point of stories is that they have acceptance criteria that can be tested for, and demo'ed at the end of a sprint (see the point on creating user value below).

How can you demo that your code base has become more habitable? Will the acceptance criterion be that the code is “just, you know, nicer”? In practice, refactoring that aims to improve existing code is done as a part of new stories. Instead of simply adding more to the spaghetti code that exists, you try to “leave the grounds better than you found”, as per Pragmatic Programmer lore. This might well be true of simple refactoring where you move code, reorganize classes, or rename things, but the really complicated cases that require rethinking of the base abstractions cannot be covered with this simple recipe.

I definitely understand the need for making software development plannable for business purposes. If the business people cannot rely on some kind of an estimate as to how much can be achieved in a certain time frame, they are navigating in the dark, and the developers are in danger of losing their jobs.

Programmers also occasionally dig deeper when they have already dug themselves into a hole, so it makes sense to set limits to stories. But there *must* be better ways to make reliable estimations of how much effort stories require.

Meeting extravaganza

Scrum meetings (aka rituals) have been among the most miserable hours of my life, and this is coming from someone with a number of visits at the Berlin foreigners’ office.

First of all, they are long. I don’t care that the meetings take place every 2 weeks if I’m going to be sitting there for three hours. They have too many attendants. Most of the stuff presented is not relevant for most people, but everyone comes because there might be something relevant, and because they have to.

The review meeting causes utterly unnecessary anxiety (*Oh my god, will my feature work?*). It’s as if the whole work of the sprint will get evaluated then and there (which in some Scrum implementations it actually is the case), and you either get the points or don’t, no matter how much thought you put into a piece of work.

The app is now faster? Who cares, I don’t get the exact response that was expected, so *no points for you*. One implicit requirement of every story is thus “should be reliably demo-able to a roomful of people”, which requires much more work than you would imagine (think payments).

In the planning meeting, you get to discuss with others about whether something is two points or five, and then actually list the things you are going to do. I presented my gripes with story points above, but in the context of planning meetings a few more sentences are in order. Why estimate stories that you are going to break down anyway? The breakdown will be a much more detailed analysis of stories, so doing that would provide a much more precise estimate.

Another thing that outright astonishes me is how little attention is paid to whether estimations are correct. This is one area where teams can learn the most, because incorrect estimates point to misunderstanding of the code base and domain, but decent review of estimates is rarely, if ever done. Tracking and estimate reviews would also enable Monte Carlo simulation of delivery dates, which sounds awesome, but is, again, rarely done.

Next up is retrospective. Frequent feedback meetings (in which the estimates are also reviewed) are actually a great idea, because the best opportunity to learn from something is right after it happened. But in Scrum, the retro is explicitly supposed to be about the Scrum process itself, not about the codebase, the technology stack or development patterns. So you have this hour for the team, and you are supposed to use it to talk about Scrum itself. Blergh.

The daily standup deserves a blog post of its own. This religious ritual has become a staple of every team in the world. Ten minutes of staring into the void, talking about what you did while no one else listens, because they were in the middle of something five minutes ago and will go back to it in another five minutes, and waiting for everyone else to finish.

I know this sounds cynical, but it is the end result of asking people to do it every freaking day. Nowadays devs are communicating on all kinds of channels (email, Slack, Github/Gitlab, ticketing system) and tracking detailed progress on some of these. What’s the point in having them stand around for another ten minutes to repeat a few standard sentences? The daily standup is in my opinion a manifestation of a significant but unspoken component of Scrum: control. The main goal of Scrum is to minimize risk and make sure the developers do not deviate from the plan. I will come back to “Scrum control mania” later.

One problem Scrum meetings share with all other meetings is that they are synchronous. For teams working remotely, this can become a serious issue, because you have to synch across continents, ending up with people attending meetings at 7 in the morning on one side of the world, and at 4 in the afternoon on the other.

This might sound like a simple scheduling problem, but synchronicity is more than that: it means cutting into people’s daily routines to force information exchange that could as well be handled otherwise.

As argued here, the agile manifesto is complicit in this meeting obsession, due to its emphasis on face-to-face communication. What I have a hard time understanding is why the ancient, simple communication form of *text* is given second seat. The truth of the matter is that, especially under the constraint of distributed teams, it’s difficult to beat text.

It is definitely true that writing well without offending others is not the simplest thing in the world, but why not educate the developers and stakeholders in this dark art? They will have to learn to communicate anyway, so you might target this asynchronous mode of communication supported by all tools out there. Text is the best means of communication, and a team that masters it will have a huge advantage. Scrum, however, does not build on text, but on meetings.

Sprint until your tongue is hanging out

Scrum is organized in units of sprints. A sprint is an iteration in which work is done, evaluated, and the process is adapted. The idea of the sprint is that the developers take on a certain amount of work, and do their best to finish it, as in, you know, they sprint. Nobody is allowed to change the acceptance criteria of the stories in the sprint, or add/remove stories. The sprint has its own backlog, which can be changed only in agreement with the team and the product owner.

I find the idea that you should get somewhere by sprinting repeatedly rather weird. As any developer will tell you, software development is a marathon, not a series of sprints. But let’s forget the semantic point for a moment, since it’s a bit too obvious, and scrum proponents could claim it’s just a convention that does not have to reflect the actual spirit.

But still, why the artificial two weeks unit? In the above mentioned guide, there is even talk of four weeks. Four weeks is a lot of time, and it is an ordinary occurrence that one or more stories become superfluous the way they were written, or other, more urgent things come into focus. If the aim is to be agile, why not accept this as the correct way to work in the first place?

In my experience, two weeks is too long for review purposes, too: it’s impossible to remember at the end of the sprint what bothered or satisfied you in the beginning. If you shorten it to one week, however, it feels like spending twice the time in the scrum rituals, although they might be shorter.

There is a more fundamental problem with the sprint idea, in my opinion. The reason software is so difficult to plan is that you discover new things about the problem at hand and your idea of a solution as you implement it. These discoveries affect not only the estimate, but also the actual path you are taking to the solution (as excellently described in this Quora answer).

The immediate work items, which constitute the head of the backlog, are the most affected by these discoveries. So essentially, a sprint is working on a frozen set of items that are most prone to change within that time frame. This is also relevant for the point made above, of assuming a too linear trajectory for software development.

Oversimplification of development process

What’s so difficult about software development? Write stories, put them on a board, split them into tasks, and then start processing them from the top to the bottom. Gather points by closing stories, pick a new story after closing one, and watch your burndown chart go down.

There are a million complications with this approach, of course. How should the teams manage dependencies among each other's' backlogs? Can I collaborate with someone, or make sure that I'm not stepping on someone else's toes?

One of the most central questions of large-scale software development alluded to above is how to rearrange work in the face of new discoveries as you are actually working; how to rebuild the ship while you're sailing, so to say. This does happen in Scrum within the sprint, and the results of the sprint flow into the next planning session, but it is not foreseen, or even taken to be possible, that the development team can rearrange work *while* it is making progress.

The Scrum coach will find fifty ways of attacking each and every one of these topics, but all of them will be in the form of *one more thing*. One more meeting, one more document, one more backlog, one more item in the definition of done.

The development process of a scrum team resembles one of those overly-pimped cars after a while: there are so many fancy bits and pieces that the actual car is not recognizable underneath anymore. The development process starts to resemble the oft mocked enterprise development process, where devs are occupied with attending meetings and filling up some documents more than anything else. Talking about the code the team is writing, and how to improve the codebase, might just be one of the meetings among others, if it at all exists.

Creating customer value

Every story in scrum has to end in customer value. The acceptance criteria have to explicitly state what the customers will derive from the results of that story, in the well-known "As a ..." format.

The idea sounds great in its simplicity, but leads to some really convoluted results when taken to the extreme (which Scrum masters have consistently told me should be done). The most obvious thing is refactoring, already mentioned above. If neither the behavior nor performance change, why even bother with refactoring? And one thing I would be ready to bet my career on is if you want to develop quality software, you should always be refactoring.

As an engineer, I care about many things that will not lead to more sales, or the customer going "It got better" in the very short run. Making the platform more reliable, understandable, aesthetically pleasing is worth spending time on, but none of this is easily expressible as delivering customer value.

For that matter, is writing a blog post delivering customer value? Will I get points for it? "As a customer, I want to read Ulas's blog post" just doesn't sound right. What about contributing to open-source software? Reading the code of an important external dependency, such as the web framework your team uses, and working on bugs or feature requests to get a better understanding was not part of any Scrum backlog I've ever seen.

One more note on refactoring, since this is a favorite topic of mine. Why is it that scrum coaches keep on saying "You should always be refactoring"? Because the assumption is that refactoring will be a few hours' work, or even shorter if it's renaming a class here and replacing a file there. These are only the most superficial cases of refactoring, however. The most difficult refactorings, incidentally, and also the ones that make the biggest difference, are these target balls of mud that need considerable effort and work to disentangle, and this is not happening "always".

It is the ideal condition to be able to do mini-refactorings, and improve code little by little, but small steps bring you nowhere in the case of these hardened balls of mud. It's of course well and dandy if you can somehow magically plan such a complicated refactoring and find a place for it in your backlog. If you can't, which is much more probable given that deep-reaching refactoring is difficult to foresee, good luck telling your product owner that you will be lost in the depths of your codebase for a while.

Scrum is not native to software

Any team that builds something can work on Scrum. This is often touted as a selling point, but it is admission of a shortcoming, in my opinion. Claiming that Scrum is generic is admitting that it is not cut for the specific nature of software development.

What is the job of a software developer? Writing code? I don't think so. I think it's inventing and customizing machine-executable abstractions, and Scrum has no facilitating aspects specifically for this kind of work. Scrum does not tell you how to organize interdependent processes that mutate while they are in flux. It doesn't tell you how to match domains to common abstractions. It doesn't tell you how to distinguish important differences from superficial ones based on context.

Of course, one can claim that this is not the job of Scrum, which is a software *management* methodology, and not a software *engineering* methodology, that it's only concerned with organizing the teams' time and workload, and anything else is the business of an engineering methodology, such as XP.

If that is the case, why the hell am *I*, the software engineer, doing most of the work — apart from the product owner, whose job *description* is doing Scrum anyway? Isn't it by definition the job of the managers, and not of the developers, to be practicing Scrum?

Shouldn't I, as a developer, be spending that whole batch of time and energy on software engineering relevant things, instead of on demoing stories, discussing the ordering of stories, and debugging the process itself? Why are the developers practicing only Scrum, and not, let's say, XP with bits of Scrum thrown in?

Another sign of the software-distant nature of Scrum is how little talk there is of an agile *codebase* in Scrum organizations. It's a *non sequitur* to think that Scrum is agile, agile teams produce agile code, ergo Scrum teams produce agile code.

Having and keeping an agile codebase is crucial to "being" agile, and is actually hard work that requires much more than only following Scrum. It is difficult to introduce processes to manage this work, however, because:

- Scrum makes claims that it is enough for design to "emerge".
- Where there is Scrum, people are reluctant to introduce even more rituals and documents.

In short: does scrum help you write good code? Does it help you achieve modularization, expression, complexity reduction? The simplest answer I have is a clear *no*.

Scrum inhibits deep understanding and innovation

This is actually my biggest gripe about Scrum. As mentioned above, in Scrum, the gods of story points per sprint reign supreme. For anything that doesn't bring in points, you need to get the permission of the product owner or scrum master or someone who has a say over them.

Refactoring, reading code, researching a topic in detail are all seen as "not working on actual story points, which is what you are paid to do". Specialization is frowned upon. Whatever technology you develop or introduce, you are not allowed to become an expert at it, because it is finishing a story that brings the points, not getting the gist of a technology or mastering an idea. These are all manifestations of the control mania of Scrum.

I recently read *Innovation: The Missing Dimension* (my review of the book), a book that focuses on an aspect of innovation that is invisible if you look at design only from a problem-solving perspective. An important part of solving a problem is finding the right problem to solve, and this cannot be treated as a problem itself. It rather requires a community (what the authors call an interpretive community) that can reformulate the given domain and create linguistic and technological tools that allow novelty.

This idea is inherent to the original agile principles in the form of individuals and interactions taking precedence over processes and tools. Scrum, however, is much closer to the problem solving approach, where analysis (breaking down a problem, and reassembling the solution) is the organizational tool.

In order for an interpretive community to emerge, an organization needs ambiguity, open-ended conversations, and alternative perceptions. In all of this, Scrum leaves to something else, whatever it is. They are not the domain of Scrum, but where there is Scrum, there is very little time and energy left for anything else.

What's more, the conditions necessary for the emergence of an interpretive community, and thus for innovation, are seen by Scrum as risk that has to be controlled and eliminated. You cannot *Scrum* innovation.

You might of course think that innovation is not necessary for you, or that it's overrated, and your company can survive without innovating. But keep in mind that the software industry is probably *the* most innovative one out there. There are new technologies every day, and the basic tools of software development go through revolutions every couple of years.

If you're not innovating, someone else who does might knock on the doors of your customers at some point. Also, innovation, in the sense of reinventing, is what software developers love to do, and is a great incentive for keeping top talent.

Summary, ideas for alternatives

So, in summary, Scrum

- wastes too much of the developers' time for management
- does not lead to good quality code
- is a control freak that does not leave room for new ideas and innovation.

Discussion on software methodologies are a bit like discussions of open-source software. The default answer to any substantial criticism is "What is your alternative?", which is pretty much the equivalent of "Why don't you submit a patch?".

Unfortunately, software management lies in the intersection of many disciplines, and is a huge field itself. My priorities as a developer lie elsewhere, namely in algorithms, programming languages, computer networks, etc. I cannot squeeze in 500-page-tomes on software management into my already crammed bookshelves.

Which won't hold me back from making probably ill-advised and rather general proposals, or at least a clarification of my expectations as a dev. First, estimations and forecasting. I don't think there is anything wrong with estimating individual stories in terms of time, and deriving a general estimate of how long a project will take from this.

The problem here is that the way stories are split and estimated is orthogonal to the way devs are working. That is, the work I, as a dev, put into organizing the backlog is not helping me in processing the backlog. If it were possible to organize and study the backlog so that this process also helps the devs, they would do it much better and more eagerly.

One way to achieve this might be putting work items through what I would call an *algebra of complexity*, i.e. an analysis of the sources of complexity in a work item and how they combine to create delays. The team could then study the backlog to locate the compositions that cause the most work and stress, and solve these knots to improve the codebase. The backlog would then resemble a network of equations, instead of a list of items, where solving one equation would simplify the others by replacing unknowns with more precise values.

The other proposal I would have is to get rid of the review, planning and standup meetings. Like, just stop doing them. There is no reason to make them so grand and rigid. You can replace most of the synchronous communication with textual communication, and create ad hoc meetings to discuss specific work items.

Instead of having sprints that are marked by these meetings, one could simply point to the backlog as a measure of work done and pending. The retrospective, on the other hand, is the only meeting in which I saw magic happen in Scrum, but it has to happen more frequently, and concentrate more on the code base, as mentioned above.

To make it short, my dream workflow would combine offline working, continuous analysis of the sources of complexity and errors, and detailed, open-ended discussion on the path on which the team is approaching the goal (or not). The correct way of building software should align the understanding that devs have of the problem and the complexity involved with the aims of the other parts of the company. ■

I don't care how well you code, understand your compensation

By ARTHUR LEVY

The scientists [and engineers] never make any money. They're always deluded into thinking that they live in a just universe that will reward them for their work and for their inventions."

As I listened to Peter Thiel's lecture, I nodded at the relevance of this point today: $585B of unicorn value is starting to falter and the inevitable fire-sales are being announced. Employees who never fully understood their equity packages are experiencing a rude awakening as investors escape safely on Preference stacks while workers get little or no return for their years of hard work.

While I am fortunate to work at a company that does a good job of explaining investor terms, not everyone is so lucky. In fact, one of the reasons I chose to work on Wall Street prior to joining a startup was to fully grasp the financing terms of the companies for which I would later work at.

My father learned that lesson the hard way. Two years ago, he called me to explain the terms of his biotech company's reverse-merger IPO. Never mind that he founded the company 20 years ago. Never mind that as a lifelong scientist entrepreneur, he had conceived new antibiotics to fight community acquired bacterial infections that will save lives for years to come.

Unfortunately, he had also imagined a "just universe" where he would be rewarded for his work and now needed me to explain what the hell had happened.

I was happy to help, but the bigger issue was the innate trust. He had never requested prior counsel to analyze the over-dilution and cram-down of his Founder shares and fight for protections.

While there may not have been a choice but to take the investment (FDA-mandated testing is expensive), it was hard to stomach that my father, an inventor of world-changing drugs, had not previously reviewed the investment terms with a fine-tooth comb. Why had he politely remained in the lab (as Chief Scientific Officer) and trusted the Board's business decisions, rather than been intimately involved in the business decisions of HIS company?

My entrepreneurial father inspired me to help build companies. But his story is emblematic of the chasm between Silicon Valley's technical and business minds.

Every week, I meet brilliant engineers and talented designers who accept option packages without understanding the first thing about the underlying equity instrument to which they are tying their fates—besides someone assuring them that they're joining "a high-growth, sexy company."

Further, when things begin to go south, many don't take the time to re-evaluate what the dilutive financings will mean for them *personally*.

I find this lack of diligence infuriating. These intellectuals—much smarter than me—imagine block chain technology for cryptocurrencies and cure diseases, but don't put in the effort to comprehend ownership percentages or what a ratchet means in a down round. Why?

Consider Good Technology, which recently sold to Blackberry for $425M, 40% of its previous valuation. While I don't know the nuances of the sale, I do know employees were advised to exercise their stock options years ago, prior to any liquidity event.

Which muppets advised them to spend hard-earned cash to exercise a non-liquid, highly volatile financial instrument? There is no world where the (relatively) small tax breaks involved justify the expected value equation, given that Good Technology was nowhere near exit.

To all the technically gifted minds out there—please talk to a financially savvy individual to avoid missteps that could impact your long-term wealth. It could prove invaluable in assessing a proffered package or the value of your current equity. I've personally helped friends at many companies better understand—and negotiate—their compensation.

At this point, you might be thinking: "To whom is this finance bro evangelizing? Does he not realize that people join startups for the thrill and excitement of working with other young smart people? It's not about the money!"

And that's fine.

But you should want to be compensated for your risk. Every equation has an expected value solution. So if you're going to accept a negative value, high-risk package, do it because you believe in the business, not because you don't appreciate the potential downside.

If the company craters, at least you'll know why your shares did too. Ask questions and seek advice—about anti-dilution provisions, about IPO protections, about liquidity preferences, about 409-A pricing.

You wouldn't ship code before QA-ing it or unveil a new design interface without peer review, so why would you believe someone who tells you, "Don't worry, we'll all make a lot of money if we do well."

What if we don't do well?

Don't be shy. This is your lifeblood. You will put in countless hours to make your company great. So talk to the CFO about the numbers—when you're hired, when it's fundraising time, and any time in between.

Take ownership of your financial future—nobody is looking out for you but yourself. Don't make the same mistake as countless others before you. *Understand your comp.* ■

Have any questions about startup compensation? Connect directly with me at arthur.r.levy@gmail.com or on Twitter or LinkedIn.

What's in a name?

Plenty, it seems, judging from the way food companies have been renaming their products to improve sales. Take for example the highly-prized Chilean Seabass, which was once known as the less appetizing Patagonian toothfish. The name change was so successful it caused the species to become overfished. And the next time you are offered some orange roughy, you might want to remember that this fish also goes by a more, *ahem*, colorful name: slimehead.

Rechristening prunes as dried plums was another stroke of genius: prunes, after all, are associated with digestion problems, while dried plums are just, you know, fruit. The awkward-sounding Chinese gooseberry is not likely to fly off the shelves, but with a catchy name like kiwi, the fruit just might. Speaking of awkwardness, is there anything worse than eating something called rape oil? Fortunately, that oil has since been renamed canola.

* FOOD BIT is where we, enthusiasts of all edibles, sneak in a fun fact about food.